Resource Control Graphs

JEAN-YVES MOYEN
LIPN – UMR 7030
CNRS – Université Paris 13
F-93430 Villetaneuse France

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

Termination is proved by finding decreases in a well-founded order on state-size, in line with other termination analyses, resulting in proofs similar in spirit to those produced by Size Change Termination analysis.

However, the size of states may also be used to measure the amount of space consumed by the program at each point of execution. This leads to an alternate characterisation of the Non Size Increasing programs, i.e. of programs that can compute without allocating new memory.

This new tool is able to encompass several existing analyses and similarities with other studies hint that even more 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 to predict and control computational resources, like space or time, that 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 a low-level language by means of Resource Control Graph, and we think that this is a generic concept from which several program properties could be checked.

Usual data flow analyses (see Nielson et al. [1999] for a detailed overview) use transfer

Author's address: J.-Y. Moyen, LIPN, Institut Galilée, 99 avenue J.B. Clément, 93430 Villetaneuse, France.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1529-3785/YY/00-0001 $5.00

functions to express how a given property is modified when following the program's execution. Then, a fixed point algorithm finds for each label a set of all possible values for the property. For example, one might be interested in which sign a given variable can take at each point. The instructions of the program give constraints on this (from one label to the next one). Iterating these constraints with a fixed point algorithm can find the set of all possible signs for the variable at each label.

Here, we want to consider each execution separately. So, when iterating the transfer function and coming back to an already treated label, instead of unifying the new constraint with the old one and iterating towards a fixed point, we will consider this as a new configuration. In the end, instead of having one set associated to each label, we will get a set of so called "walks", each associating one value to each occurrence of each label. For example, a first walk can tell that if starting with a positive value at a given label, the variable will stay positive, but another walk tells that if starting with a negative value, the variable may become positive. In this case, the fixed point algorithm will build the set $\{+, -\}$ for each label.

Of course, we then need a way to study this set of walks and find common properties on them that tell something about the program.

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 [1999].

There are several approaches that try to solve this problem. The first protection mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could itself cause memory leak or other problems. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a lower bound on the memory usage while static analysis gives an upper bound. The gap between both bounds is of some value in practice. 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 that 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 work 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 foundation of recent research on describing broad classes of programs that run within some specified amount of time or space. Examples include Hoffmann as well as Niggl and Wunderlich [2006], Amadio et al. [2004] and Bonfante et al. [2007].

We believe that our Resource Control Graphs will be able to encompass several, or even all, of these analyses and express them in a common framework. 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.

Following Jones [1997], we call a *program property* a subset of all programs (for a given, yet unspecified, language) and we say that a property A is *extensional* if for all programs p, q computing the same function, $p \in A \Leftrightarrow q \in A$. That is, an extensional property is shared by all programs computing the same function.

On the other hand, properties not shared by programs computing the same function are called *intensional*. A typical extensional property is termination. Indeed, all programs computing the same function must terminate on the same inputs (where the function is defined). A typical intensional property is time complexity. Indeed, two programs with different complexities can compute the same function; for example, insertion sort computes the sorting function in time $O(n^2)$ while merge sort computes it in time $O(n \log(n))$.

Classical complexity theory focuses on functions or problems and extensional properties. It defines complexity classes (such as LOGSPACE, PTIME) as classes of *problems* and not classes of algorithms and studies complexity of problems (SAT, QBF, ...) and relationship between classes of problems ("Is P different from NP ?", ...) Here, we want to consider *intensional* complexity, that is try to understand why a given algorithm is more efficient than another to compute the same function, and we want to study classes of *algorithms* rather than classes of problems or functions.

When studying the complexity of functions, one deals with extensional properties; however, the complexity of algorithms is an intensional property. Consider for example the class $\mathcal{C}$ of *functions* computable in time $O(n \log(n))$. This is a class of functions, hence not a program property. Consider now the corresponding program property: A is the set of all programs computing a function from $\mathcal{C}$. Obviously, the merge sort algorithm is in A. But the insertion sort algorithm, with complexity $O(n^2)$ is *also* in A. Indeed, the function computed by the insertion sort, the sorting function, is in $\mathcal{C}$ because there exists a $O(n \log(n))$ program to compute it.

On the other hand, the complexity of algorithms is an intensional property. Consider the program property B, the set of all programs whose complexity is $O(n \log(n))$. Now, the merge sort belongs to B because its complexity is $O(n \log(n))$ but the insertion sort does not belong to B because its complexity is $O(n^2)$. We no longer consider only the computed function, but also the algorithm used to compute it. For this reason, "bad" programs are discarded.

A typical goal of intensional study would be to have a criterion to decide whether a given program computes in polynomial time or not.

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.

However, several very successful works do exist for studying both extensional properties (like termination) or intensional ones (like time or space complexity of programs). 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 (usually complexity of algorithms), two different kinds of approaches exist. The first one consists in restricting the syntax of programs so that any program necessarily has the wanted property. The work on primitive recursive functions, where the recurrence schemata are restricted to only primitive recursion, falls into this category. 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 each *function* in the class (of functions) considered can be computed by an algorithm with the property considered (*e.g.*, a function is in PTIME if and only if it can be defined by bounded primitive recursion (Cobham)). Unfortunately, *intensionality* is not their main concern: these methods usually do not capture natural algorithms [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 one 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 follow a different direction: Let the programmer write whatever he wants, but the analysis is not guaranteed to work.

Since any program can *a priori* be given to the analysis, decidability is generally achieved by loosening the semantics during analysis. That is, one will consider *more* than all the executions a program can have.This approach is more recent but has already some very successful results such as the Size Change Termination [Lee et al. 2001] or the mwp-polynomials of Kristiansen and Jones [2005].

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 will not 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.

Resource Control Graphs are intended to live within this second kind of analysis. Hence, the toy language used as an example is Turing-complete and will not be restricted.

1.3 Outline

Section 2 introduces the stack machines, used everywhere in the remainder of this paper, as a simple yet powerful programming language. Section 3 describes 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 4 immediately shows how this can be used in order to detect Non Size Increasing programs. Section 5 presents Vector Addition Systems with States that are generalised into Resource Systems with States in Section 6. They form the backbone of the Resource Control Graphs. Section 7 presents the tool itself and explains how to build a Resource Control Graph for a program and how it can be used to study the program. Section 8 shows application of RCGs in building termination proofs similar to the Size Change Termination principle. Finally, Section 9 discusses how matrix algebra could be used in program analyses, leading to several possible further developments.

1.4 Notations

In a directed graph[1] $G = (S, A)$, will write $s\xrightarrow{a}s'$ to say that a is an edge between s and s'. 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 passing 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 $\to$.

A partial order $\prec$ is a *well partial* order if there are no infinite decreasing sequence and no infinite anti-chain. That is, for every infinite sequence $x_1, \cdots, x_n, \dots$ there are indexes $i < j$ such that $x_i \preceq x_j$. This means 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 well-founded but is not a well partial order since the sequence of all prime numbers is an infinite sequence of pairwise incomparable elements.

The set of integers (positive and negative) is $\mathbb{Z}$, and $\mathbb{N}$ is the set of integers ≥ 0. When working with infinity, $\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$; that is, we do not need $-\infty$ here. When working with vectors of $\mathbb{Z}^k$, $\leq$ denotes the component-wise partial order. That is $a \leq b$ if and only if $a_i \leq b_i$ for all $1 \leq i \leq k$. This is a well partial order on $\mathbb{N}^k$.

If w is a word and a a letter, we note $a.w$ the word that begins with the letter a and ends with w. Similarly, if w is a stack and a a value, $a.w$ denotes the stack whose top is a and whose tail is w.

2. STACK MACHINES

2.1 Syntax

A stack machine consists 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 *(Stack machine).* Stack machines are defined by the following grammar[2]:

(Alphabet)	Σ		finite set of symbols
(Programs)		$p ::=$	$\mathtt{lbl}_1 : \mathtt{i}_1; \dots; \mathtt{lbl}_n : \mathtt{i}_n;$
(Instructions)	$\mathcal{I} \ni$	$\mathtt{i} ::=$	$\mathtt{if}$ (test) $\mathtt{then\ goto\ lbl}_0\ \mathtt{else\ goto\ lbl}_1 \mid \mathbf{r} := \mathtt{pop}(\mathbf{stk}) \mid \mathtt{push}(\mathbf{r}, \mathbf{stk}) \mid \mathbf{r} := \mathtt{op}(\mathbf{r}_1, \cdots, \mathbf{r}_k) \mid \mathtt{end}$
(Labels)	$\mathcal{L} \ni$	$\mathtt{lbl}$	finite set of labels
(Registers)	$\mathcal{R} \ni$	$\mathbf{r}$	finite set of registers
(Stacks)	$\mathcal{S} \ni$	$\mathbf{stk}$	finite set of stacks
(Operators)	$\mathcal{O} \ni$	$\mathtt{op}$	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 $\mathtt{next} : \mathcal{L} \to \mathcal{L}$ such that $\mathtt{next}(\mathtt{lbl}_i) = \mathtt{lbl}_{i+1}$ and a mapping $\iota : \mathcal{L} \to \mathcal{I}$ such that $\iota(\mathtt{lbl}_k) = \mathtt{i}_k$. The `pop` operation removes the top symbol of a stack and put it in a register. The `push` operation copies the symbol in the register onto the top

[1]We will use $s \in S$ to designate vertices and $a \in A$ to designate edges. The choice of using French initials ("Sommet" and "Arête") rather than the usual (V, E) is done to avoid confusion between vertices and the valuations introduced later.
[2]We use a bold face font for registers and stacks and a typewriter font for instructions

of the stack. The ιif-instruction gives 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 semantic function $[\![\texttt{op}]\!]$.

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

The syntax $\texttt{lbl} : \texttt{if}$ (test) $\texttt{then goto lbl}_0$ can be used as a shorthand for $\texttt{lbl} : \texttt{if}$ (test) $\texttt{then goto lbl}_0$ $\texttt{else goto next}(\texttt{lbl})$. Similarly, we can abbreviate $\texttt{if}$ true $\texttt{then goto lbl}$ as $\texttt{goto lbl}$, that is an unconditional jump to a given label. The kinds of tests allowed are not specified here. Of course, tests must be computable (for obvious reasons) in constant time and space, so that they do not play an important part when dealing with complexity properties. Comparisons between letters of the alphabet (*e.g.* $\leq$ if they are integers) are typical tests that can be used.

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 a usual counter machine [Shepherdson and Sturgis 1963].

Example 2.2. The following program reverses a list in stack **l** and put the result in stack **l**′ (assuming **l**′ is empty at the beginning of the execution). It uses register **a** to store intermediate letters. The empty stack is denoted $[]$.

```
  0 : if l = [] then goto end;   |     3 : goto 0;
  1 : a := pop(l);               |   end : end;
  2 : push(a, l′);               |
```

2.2 Semantics

Definition 2.3 *(Stores).* A *store* is a function σ assigning a symbol (letter of the alphabet) to each register and a finite string in Σ^* to each stack. Store update is denoted $\sigma\{x \leftarrow v\}$.

Definition 2.4 *(States).* Let p be a stack program. A *state* of p is a pair $\theta = \langle \texttt{IP}, \sigma\rangle$ where the *Instruction Pointer* $\texttt{IP}$ 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 the two.

Definition 2.5 *(Executions).* The operational semantics of Figure 1 defines a relation[3] $p \vdash \theta \xrightarrow{\texttt{i}} \theta'$.

An *execution* of a program p is a sequence (finite or not) $p \vdash \theta_0 \xrightarrow{\texttt{i}_1} \theta_1 \xrightarrow{\texttt{i}_2} \ldots \xrightarrow{\texttt{i}_n} \theta_n \ldots$

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

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

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

[3]Notice that the label $\texttt{i}$ on the edge is technically not an instruction since for tests we also keep the information of which branch is taken.

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

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

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

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

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

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

Fig. 1. Small steps semantics

Definition 2.6 *(Traces).* The *trace* of an execution $p \vdash \theta_0 \xrightarrow{\mathtt{i}_1} \theta_1 \xrightarrow{\mathtt{i}_2} \ldots \xrightarrow{\mathtt{i}_n} \theta_n \ldots$ is the instructions sequence $\mathtt{i}_1 \ldots \mathtt{i}_n \ldots$

Definition 2.7 *(Length).* Let $\theta = \langle \mathtt{IP}, \sigma\rangle$ be a state. Its *length* $|\theta|$ is the sum of the number of elements in each stack[4]. That is:

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

The length of a state corresponds to the usual notion of space consumption. Since there is a fixed number of registers and each can only store a finite number of different values, the space need to 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 an execution is the length of the corresponding sequence (possibly infinite for non-terminating programs). Let f be an increasing function from the non-negative integers to the non-negative integers. We say that the *running- time* of a program is bounded by f if the time usage of each execution is bounded by $f(|\theta|)$ where θ is the first state of the execution.

The *space usage* of an execution is the least upper bound of the length of a state in it. Let f be an increasing function from the non-negative integers to the non-negative integers.

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

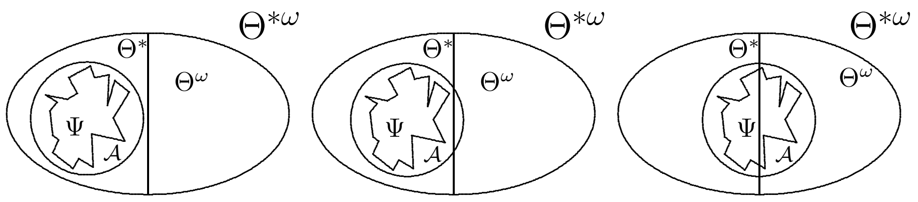

Fig. 2. Sequences of states, executions and admissible sequences

We say that the *running- space* of a program is bounded by f if the space usage of each execution is bounded by $f(|\theta|)$ where θ is the first state of the execution.

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

FPTIME denotes the set of functions computable in polynomial time by a stack machine, that is $f \in$ FPTIME if and only if $f \in T(P)$ for some polynomial P.

If we want to define classes such as LOGSPACE, then we must, as usual, use some read-only stacks that can only be poped but not pushed and some write-only output stacks. These play no role when computing the length of a state.

2.3 Turing Machines

Stack machines are Turing complete. We quickly describe here the straightforward ways to simulate one model by the other.

Simulating a stack machine with k stacks can be done by a Turing machine with $k+1$ tapes. The first k tapes each store one of the stacks while the last one store all the registers. The different instructions are represented by the states of the machine.

Simulating a TM with a single tape can be done by using two stacks to represent the tape in an usual way (each stack represent half of the tape). Registers are used to store the current state as well as the scanned symbol.

3. A TASTE OF RCG

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

3.1 Admissible sequences

Consider an execution of a program. It can be described as a sequence of states. Clearly, not all sequences of states describe an execution. So we have a set of executions, Ψ, which is a subset of the set of all sequences of states (finite or infinite), $\Theta^{*\omega}$.

The undecidability results entail that, given a program, it is impossible to say whether the set Ψ of executions, and the set Θ^ω of infinite sequences of states, are disjoint. So, the idea here is to find a set $\mathcal{A}$ of *admissible* sequences that 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 fortiori*, there are no infinite executions of the program; but if the intersection is not empty, then we cannot decide whether this is due to some non-terminating execution of the program or to some of the sequences added for the sake of the analysis. This means that depending on the machine considered and the way $\mathcal{A}$ is built, we can be in three different situations as depicted in Figure 2. We build $\mathcal{A} \supset \Psi$ such that $\mathcal{A} \bigcap \Theta^\omega$ is decidable. If it is empty, then the program uniformly terminates; otherwise, we cannot say anything. Of course, the undecidability theorem implies that if we require $\mathcal{A}$ to be recursive (or at least recursively separable from Θ^ω), then there will necessarily be some programs for which the situation will be the one in the middle (in Figure 2), i.e. the program uniformly terminates, but we cannot determine that it does.

One conceptually simple way to represent all the possible executions (and only these), is to build a *state-transition graph*. This is a directed graph where each vertex is a state of the program and there is an edge between two vertices if and only if it is possible to go from one state to the other with a single step of the operational semantics. Of course, since there are infinitely many different stores, there are infinitely many possible states and the graph is infinite.

3.2 The folding trick

Using the state-transition graph to represent executions is not convenient since handling an infinite graph can be tedious. To circumvent this, we must look into states and decompose them.

A state is actually a pair of one label and one store. The label corresponds to the *control* of the program while the store represents *memory*. A first try to get rid of the infinite state-transition graph is then to only consider the control part of each state.

Thus, there will only be finitely many different nodes in the graph (since there are only finitely many different labels). By identifying all states bearing the same label, it becomes possible to "fold" the infinite state-transition graph into a finite graph, called the *Control Flow Graph* (CFG) of the program. The CFG is a usual tool for program analyses and transformations and can directly be built from the program.

Definition 3.1 *(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 $\iota(\texttt{lbl})$.

Vertices and edges are named after, respectively, the label or instruction[5] they represent. No distinction is made between the vertex and the label or the edge and the instruction as long as the context is clear.

Example 3.2. The CFG of the reverse program is displayed on Figure 3.

With state-transition graphs, there was a one-to-one correspondence between executions of the program and (maximal) paths in the graph. This is no longer true with Control Flow Graphs. Now, to each execution corresponds a path (finite or infinite) in the CFG. The converse, however, is not true. There are paths in the CFG that correspond to no execution.

[5] Again, since the two branches of tests are separated, some edges do not correspond exactly to an instruction of the program. We will nonetheless continue to call these "instructions".

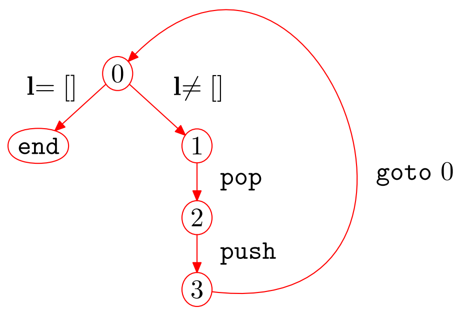

Fig. 3. CFG of the reverse program.

Let $\mathcal{P}$ be the set of paths in the CFG. Since we can associate a path to each execution, we can say that $\mathcal{P}$ is a superset of Ψ.

This leads to a first try at building a set of admissible[6] sequences by choosing $\mathcal{A} = \mathcal{P}$.

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 loops uniformly terminates*".

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

3.3 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 $[\![\mathtt{i}]\!]$ such that for all states θ, θ' for which $p \vdash \theta = \langle \mathtt{IP}, \sigma \rangle \xrightarrow{\mathtt{i}} \langle \mathtt{IP}', \sigma' \rangle = \theta'$, we have $\sigma' = [\![\mathtt{i}]\!](\sigma)$.

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 $[\![\mathtt{i}]\!]$ of the edge just followed.

The only case where the CFG has out-degree greater than 1 is for tests. In order to prevent the wrong branch to be taken, the semantics function $[\![(\text{test})_{\text{true}}]\!]$ can be a partial function only defined for stores where the test is true (and conversely for the false branch of tests).

But if we do this exactly that way, then there will be a bijection between the executions and the walks and everything will stay undecidable.

So the idea at this point is to keep both branches of the test possible, that is more or less replacing a deterministic test by a non-deterministic choice between the two outcomes. This leads to a set of walks bigger than the set of executions but, hopefully, recursively separable from the set of infinite sequences of states.

4. MONITORING SPACE USAGE

In order to illustrate the ideas of the previous Section, we introduce here the notion of Resource Control Graph for the specific case of monitoring space usage. In Section 7, this

[6] $\mathcal{P}$ is indeed recursive. By adapting Lemma 5.13, it is possible to show that $\mathcal{P}$ is an omega-regular language and hence can be recognised by a Büchi's automaton.

notion will be fully generalised to define Resource Control Graphs.

4.1 Space Resource Control Graphs

Definition 4.1 *(Weight).* For each instruction `i`, we define a *weight* $k_{\mathtt{i}}$ 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 4.2. *For all states θ such that $p \vdash \theta \xrightarrow{i} \theta'$, we have $|\theta'| = |\theta| + k_{\mathtt{i}}$.*

It is important here that both θ and θ' are states. Indeed, this means that when an error occurs ($\perp$), nothing is said about the weight of the instruction causing the error.

Definition 4.3 *(Space Resource Control Graph).* Let p be a program. Its Space Resource Control Graph (Space-RCG) is a weighted directed graph G such that:

—G is the Control Flow Graph of p.
—For each edge `i`, the weight $\omega(\mathtt{i})$ is $k_{\mathtt{i}}$.

Definition 4.4 *(Configurations, walks).* A *configuration* is a pair $\eta = (s, v)$ where $s \in S$ is a vertex and $v \in \mathbb{Z}$ is the *valuation*. A configuration is *admissible* if and only if $v \in \mathbb{N}$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (s_n, v_n) \xrightarrow{a_{n+1}} \ldots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \ldots$ and for all $i > 0$, $v_i = v_{i-1} + \omega(a_i)$. A walk is *admissible* if all configurations in it are admissible.

Definition 4.5 *(Traces).* The *trace* of a walk is the sequence of all edges followed by the walk, in order.

PROPOSITION 4.6. *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 in G, $(\mathtt{IP}_1, |\theta_1|) \to \ldots \to (\mathtt{IP}_n, |\theta_n|)$, with the same trace t.*

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

4.2 Characterisation of Space usage

THEOREM 4.7. *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 initial state $\theta_0 = \langle \mathtt{IP}_0, \sigma_0 \rangle$ with execution $p \vdash \theta_0 \xrightarrow{*} \theta_n$, the trace of the execution is also the trace of an admissible walk $(\mathtt{IP}_0, |\theta_0|) \to (\mathtt{IP}_1, v_1) \to \ldots \to (\mathtt{IP}_n, v_n)$ and for each k, $v_k \leq f(|\theta_0|)$.

PROOF. Proposition 4.6 tells us that $v_k = |\theta_k|$. Then, both implications hold by definition of space usage. □

Definition 4.8 *(Resource awareness).* A Space-RCG is *f-resource aware* if for any admissible walk $(s_0, v_0) \xrightarrow{*} (s_n, v_n)$, $v_n \leq f(v_0)$.

Non-admissible walks are not taken into account because they never correspond to real executions of the program.

COROLLARY 4.9. *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 that do not correspond to any real execution.

4.3 Non Size Increasingness

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

Hofmann detects Non Size Increasing programs in a typed functional language by adding a special type, $\diamond$, which can be seen as the type of pointers to free memory. Here, the valuations of Space RCG will play exactly the same role as Hofmann's $\diamond$, that is managing the memory freed by previous de-allocation and reuse it rather than re-allocating new memory.

Even if this work was inspired by Hofmann's, there is currently no explicit link or equivalence Theorem between the programs detected by one or the other.

Definition 4.10 *(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 that can be computed by Non Size Increasing programs. That is, $NSI = \bigcup_\alpha S(\lambda x.x + \alpha)$.

PROPOSITION 4.11. *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 4.7. □

THEOREM 4.12. *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 $(s_0, v_0)\overset{*}{\to}(s_n, v_n)$ 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, then it can be followed infinitely many times and provides an admissible walk with unbounded valuations. □

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 (and 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 4.13. The Space-RCG of the reverse program (from Example 2.2) is displayed on Figure 4. Since it contains no cycle of strictly positive weight, the program is Non Size Increasing. Moreover, since the maximum weight of any path is 1, it can be computed in space $\lambda x.x + 1$, that is the constant α is 1 for this program.

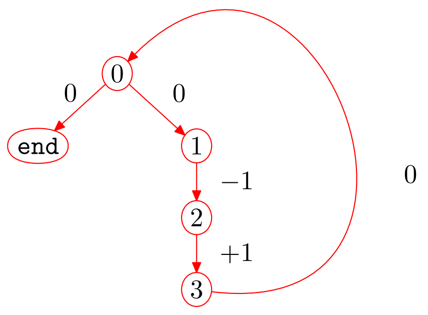

Fig. 4. Space-RCG of the reverse program.

Actually, the reverse program can be computed in space $\lambda x.x$. This is not detected because we consider here all paths and not only paths starting from 0 (the initial node, corresponding to the first label of the program). This could be improved, and should be for any practical use, but the sharp bound is not needed in our theoretical framework.

This result, however, lacks an intensionality statement (how many of all NSI programs are caught?) or even an extensional completeness one (does there exist functions in NSI that cannot be captured by such a program?) Of course, the class of all *programs* that are Non Size Increasing is undecidable. This means that intensionality statements are hard to achieve. However, we can reach an extensional completeness one.

Definition 4.14 *(Normalising programs).* Let p be a program and α be a constant. We define program $\widetilde{p}_\alpha$ as follows:

—There is an extra stack **mem** (empty at start) and an extra symbol $\diamond$.

—The α first instructions of $\widetilde{p}_\alpha$ are `push`$(\diamond, \textbf{mem})$. That is, $\widetilde{p}_\alpha$ starts by pushing α copies of $\diamond$ on **mem**.

—The following instructions of $\widetilde{p}_\alpha$ are the instructions of p, except that each `push` is followed by a `pop`(**mem**) and each `pop` is preceded by a `push`$(\diamond, \textbf{mem})$.

PROPOSITION 4.15. *If p runs in space $\lambda x.x + \alpha$, then $\widetilde{p}_\alpha$ computes the same function as p.*

PROOF. As long as they do not cause runtime error (*i.e.* `pop`ing an empty stack), the added instructions do not interfere with the computed function because they only act on the new stack **mem**.

The only runtime error that **mem** can cause would be if one tries to `pop` it when its empty. However, if this happens, then a `push` on a non-**mem** stack must have happened just before while **mem** was empty. In the state just reached by this `push`, the sum of the lengths of the non-**mem** stacks would be $x+\alpha+1$ where x is the length of the initial state. This contradicts the fact that p runs in space $\lambda x.x + \alpha$. $\square$

Notice also that such a normalisation could be made for a program running in space $f(x)$ for any computable function f. However, in that case the simulation would require to compute $f(x)$ from the input and then push sufficiently many $\diamond$s. This would be quite tricky to do and require control over the space used to compute $f(x)$. Hence, adapting these

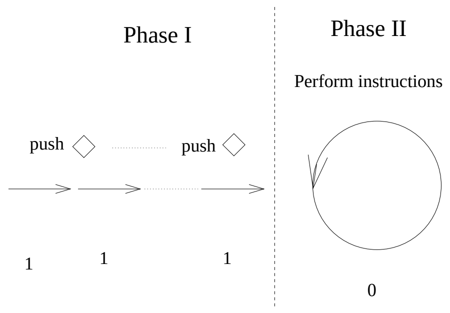

Fig. 5. Space-RCG of a normalised program.

ideas to classes of programs defined by their space complexity other than NSI cannot be done in a straightforward way and caution must be exerted.

LEMMA 4.16. *Let p be a program. The Space-RCG of $\widetilde{p}_\alpha$ has no cycle of strictly positive weight.*

PROOF. By construction of $\widetilde{p}_\alpha$. The first part (pushing α copies of $\diamond$ on **mem**) is done without any cycles in the control flow, by actually using α copies of the `push` instruction. The second part (simulating p) has each `push` paired with a `pop` (on another stack) and hence cannot generate paths of weight other than 0 or 1, and, especially, no cycle of weight different from 0. □

THEOREM 4.17 (EXTENSIONAL COMPLETENESS). *Let f be a function in* NSI. *There exists a program p computing f whose Space-RCG has no cycle of strictly positive weight.*

PROOF. Since f is in NSI, it is computed by a program q running in space $\lambda x.x + \alpha$ for some α. By Proposition 4.15 and Lemma 4.16, $p = \widetilde{q}_\alpha$ also computes f and has a Space-RCG without strictly positive cycle. □

This result means that this characterisation of NSI by programs whose Space-RCG have no cycle of strictly positive weight is extensionally complete: each function in NSI can be computed by a program that fits into the characterisation (that is, whose Space-RCG is $\lambda x.x + \alpha$-resource aware). Of course, intentional completeness (capturing all Non Size Increasing programs) is far from reached (but is unreachable with a decidable algorithm): there exist Non Size Increasing programs whose Space-RCG has cycle of positive weight (but, due to extensional completeness, the functions computed by these programs can also be computed by *another* program whose Space-RCG has no cycle of strictly positive weight).

Example 4.18. The following two programs compute the same function, namely pushing five 0s onto stack **l**. The leftmost one uses a loop whose number of iterations is fixed inside the program by the assignment at label 0, while the rightmost one uses five copies of `push`.

Since value of variables is not taken into account by the analysis, the Space-RCG of the first program will have a loop of strictly positive weight corresponding to the loop in the program. Hence, this first program is Non Size Increasing but not detected by the characterisation. This means that the characterisation is not *intensionally complete*: some programs have the wanted property but are left out. This is, obviously, an unwanted fact but nonetheless inevitable because the set of Non Size Increasing programs is undecidable.

However, because of the extensional completeness, there must exist programs computing the same function that fit into the characterisation. The second program is an example of such a program. Indeed, its Control Flow Graph does not contain any cycles, hence, the Space-RCG does not have cycles either (and so, no cycle of positive weight).

```
  0 : a := 5;
  1 : if a = 0 then goto end;
  2 : push(0, l);
  3 : a := a − 1;
  4 : goto 1;
end : end
```

```
  0 : push(0, l);
  1 : push(0, l);
  2 : push(0, l);
  3 : push(0, l);
  4 : push(0, l);
end : end
```

If one applies the normalisation of Definition 4.14 to the first program, one will obtain:

```
  0 : push(◇, mem);
  1 : push(◇, mem);
  2 : push(◇, mem);
  3 : push(◇, mem);
  4 : push(◇, mem);
  5 : a = 5;
  6 : if a = 0 then goto end;
  7 : pop(mem);
  8 : push(0, l);
  9 : a := a − 1;
 10 : goto 6;
end : end;
```

This program computes the same function. The first part (in the leftmost column) allocates a constant amount of memory on a "free memory" stack. Since the amount of memory needed (the constant α in the Lemmas and Theorems above) is known, this can be done with no loops by using several copies of `push`. Then, the program mimics the loop of the first program. However, before allocating any new memory (that is, before performing any `push`), it starts by removing some free memory from **mem** (with a `pop`). This ensures that no cycle in the Space-RCG will have a strictly positive weight.

4.4 Linear Space

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

Definition 4.19. FLINSPACE denotes the set of functions computable in linear space by a stack machine, that if $f \in$ FLINSPACE if and only if $f \in S(l)$ for some linear function $l : x \mapsto \beta x + \alpha$.

The idea is quite easy: since we are allowed to use a factor of β 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 as *input stacks* and the others must be initially empty. Then, a `pop` operation on 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 `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 `push` operation on them.

Notice that any program can be turned into such a program by having twice as many stacks (one input and one work for each) and starting by copying all the input stacks into the corresponding working stacks and then only dealing 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 *β-size*. Globally, we will use size to denote some kind of measure on states that is used by the RCG for analysis. The terminology is close to the one used for 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 the usual ordering on $\mathbb{N}$ as size ordering, even if the integers are all 32 bits integers, thus taking exactly the same space in memory.

Definition 4.20 *(Extended stack machines).* An *extended stack machine* is a stack machine with the following modification:

There are two disjoint sets of stacks, $\mathcal{S}_i$ is the set of *input stacks* and $\mathcal{S}_w$ is the set of *working stacks*. There are two instructions $\texttt{pop}_i$ and $\texttt{pop}_w$ depending on whether an input or working stack is considered but only one $\texttt{push} = \texttt{push}_w$ instruction, that is it is impossible to `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:

$$||\theta||_\beta = \beta \sum_{\mathbf{stk}_i \in \mathcal{S}_i} |\mathbf{stk}_i| + \sum_{\mathbf{stk}_w \in \mathcal{S}_w} |\mathbf{stk}_w|$$

The *weight* of $\texttt{pop}_i$ is $-\beta$, the weight of $\texttt{pop}_w$ is -1, the weight of `push` is $+1$. The weight of any other instruction is 0.

The β-Space RCG is built as the Space-RCG: the underlying graph is the control flow graph and the weight of each edge is the weight of the corresponding instruction.

Proposition 4.6 becomes:

PROPOSITION 4.21. *Let p be a program, G_β be its β-Space RCG and $p \vdash \theta_1 = \langle \mathit{IP}_1, \sigma_1\rangle \to \ldots \to \theta_n = \langle \mathit{IP}_n, \sigma_n\rangle$ be an execution with trace t, then there is an admissible walk $(\mathit{IP}_1, ||\theta_1||_\beta) \to \ldots \to (\mathit{IP}_n, ||\theta_n||_\beta)$ with the same trace t.*

Then, adapting Theorem 4.7 and Theorem 4.12, we have:

PROPOSITION 4.22. *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 $p \in S(\lambda x.\beta x + \alpha)$.*

THEOREM 4.23. *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 4.24. *Let p be a program. If there exists β such that its β-Space RCG contains no cycle of strictly positive weight, then p computes a function in* FLINSPACE.

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

Also for FLINSPACE, the normalisation process of programs can be performed. The first phase of the normalised program consists in first pushing onto **mem** α copies of $\diamond$, then

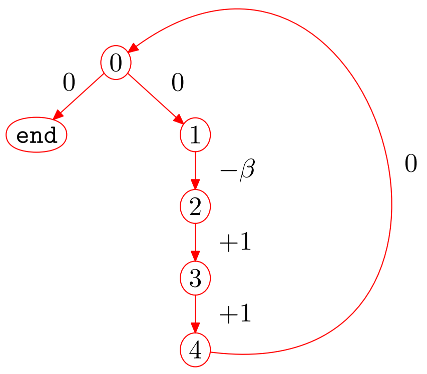

Fig. 6. β-Space RCG of the double-reverse program.

repeatedly copying each input stack onto the corresponding working stack and each time a symbol is copied, $\beta - 1$ new $\Diamond$s are pushed onto **mem** (so that the global space usage from now on is always $\beta x + \alpha$). This means that here also the characterisation is extensionally complete: for each FLINSPACE function, there exists one program computing it that fits into the characterisation.

Example 4.25. The following program "double-reverses" a list. It is similar to the reverse program but each element is present twice in the result. The list **l** is an input stack (and hence cannot be pushed) while **l′** is a working stack.

```
0 : if l = [] then goto end;     |    3 : push_w(a, l′);
1 : a := pop_i(l);               |    4 : goto 0;
2 : push_w(a, l′);               |  end : end;
```

Its β-Space RCG is displayed on Figure 6. Since it contains no cycle of strictly positive weight if $\beta \geq 2$, the program is in LINSPACE. More precisely, it can be computed in space $\lambda x.2x$

5. VECTOR ADDITION SYSTEM WITH STATES

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

5.1 Definitions

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.

A *configuration* is a pair $\eta = (s, v)$ 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 $(s_0, v_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (s_n, v_n)$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ and for all $i > 0$, $v_i = v_{i-1} + \omega(a_i)$. A walk is *admissible* if all configurations in it are admissible.

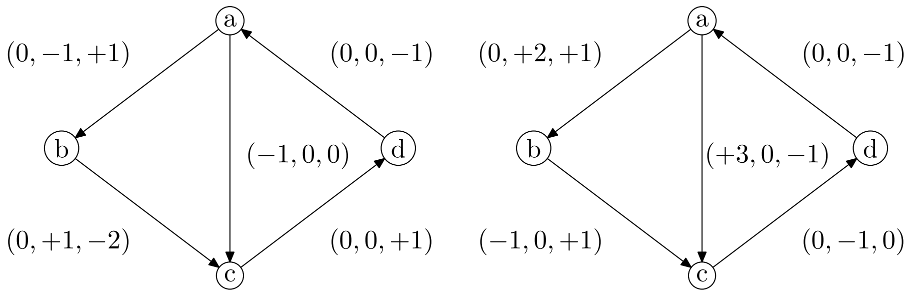

Fig. 7. Two VASS

We say that the path $a_1 \ldots a_n$ 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 $\eta \to \eta'$ if there exists an edge a such that $\eta \xrightarrow{a} \eta'$ and $\xrightarrow{+}$, $\xrightarrow{*}$ for the closures.

Definition 5.2 *(Weight of a path).* Let V be a VASS and $a_1 \ldots a_n$ be a path in it. The weight of edges is extended to paths canonically: $\omega(a_1 \ldots a_n) = \sum \omega(a_i)$. This means that ω is a morphism between $(A, \cdot)$ (the free monoid generated by the edges) and $(\mathbb{Z}^k, +)$.

Example 5.3. Figure 7 displays two VASS. More formally, the first one should be described as a graph $G = (S, A)$ with:

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

LEMMA 5.4. *Let V be a VASS and $a_1 \ldots a_n$ be a finite path in it. There exists a valuation v_0 such that for $0 \le i \le n$, $v_0 + \omega(a_1 \ldots a_i) \in \mathbb{N}^k$.*

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

PROOF. Because the path is finite, the jth component of $\omega(a_1 \ldots a_i)$ is bounded from below by some 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. □

Example 5.5. Let us consider the first VASS of Figure 7 and the path $a \xrightarrow{a_1} b \xrightarrow{a_3} c$. This path has weight $(0, 0, -1)$. If we consider the initial valuation $(0, 0, 0)$ and the corresponding walk $(a, (0, 0, 0)) \xrightarrow{a_1} (b, (0, -1, 1)) \xrightarrow{a_3} (c, (0, 0, -1))$, this walk is not admissible because the second and third valuations have a strictly negative coefficient. However, following the same path, it is always possible to take a “big enough” initial valuation in order to get an admissible walk. Here, the walk $(a, (0, 1, 1)) \xrightarrow{a_1} (b, (0, 0, 2)) \xrightarrow{a_3} (c, (0, 1, 0))$ follows the same path and is admissible.

LEMMA 5.6. *Let $(s_0, v_0) \to \ldots \to (s_n, v_n)$ be an admissible walk in a VASS. Then, for all $v'_0 \geq v_0$ (component-wise comparison), $(s_0, v'_0) \to \ldots \to (s_n, v'_n)$ is an admissible walk (following the same path).*

PROOF. By monotonicity of the addition. □

Example 5.7. Continuing the previous example, this means that any valuation larger (component-wisely) than $(0, 1, 1)$ leads to an admissible walk when following the same path. So, among other, the walk starting at $(a, (17, 14, 42))$ and following edges a_1 and a_3 is admissible.

Definition 5.8 *(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 5.9. *A VASS is* not *uniformly terminating if and only if there exists a cycle whose weight is in $\mathbb{N}^k$ (that is, is non-negative with respect to each component).*

PROOF. If such a cycle exists, starting and ending at vertex s, then by Lemma 5.4 there exists v_0 such that the walk starting at (s, v_0) and following the cycle is admissible. After following the cycle once, the configuration (s, v_1) is reached. Since the weight of the cycle is non-negative, $v_1 \geq v_0$. Then, by Lemma 5.6 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 $(s_0, v_0) \to \ldots \to (s_n, v_n) \to \ldots$ 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'_l, v'_l) be the occurrences of the corresponding configurations in the walk. Since the component-wise order over vectors of $\mathbb{N}^k$ 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 non-negative weight. □

Example 5.10. The second VASS of Figure 7 is not uniformly terminating. Indeed, if we consider the cycle $C = a_1a_3a_4a_5a_1a_3a_4a_5a_2a_4a_5$, it has weight $(1, 1, 0)$. By Lemma 5.4, there exists a valuation $v = (m, n, p)$ such that the walk which starts at $(a, (m, n, p))$ and following this cycle is admissible (it is sufficient to choose $(m, n, p) \geq (2, 0, 0)$). After following the cycle once, the configuration is $(a, (m + 1, n + 1, p))$ from which the cycle can be followed once again, thus reaching $(a, (m+2, n+2, p))$. Repeating this leads to an infinite admissible walk.

5.2 Decidability of the uniform termination

Definition 5.11 *(Linear parts, semi-linear parts).* Let $(M, +)$ be a commutative monoid[7]. A *linear part* of M is a subset of the form $v + V^*$ where $v \in M$ and V is a finite subset of M. That is, if $V = \{v_1, \cdots, v_p\}$, a linear part can be expressed as:

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

[7]Recall that a commutative monoid is a monoid (a set with an associative internal operation admitting a neutral element) whose operation is commutative. A typical example of commutative monoid is $(\mathbb{N}, \times)$ (inverse for each element is not needed).

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

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

LEMMA 5.12. *In a commutative monoid, semi-linear parts are exactly the rational parts.*

PROOF. Semi-linear parts are 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 *.

Semi-linear parts contains finite part and are closed under union by definition. Closure under sum is obtained because $(a + A^*) + (b + B^*) = (a + b) + (A \bigcup B)^*$ and sum is distributive over union ($(A \bigcup B) + C = (A + C) \bigcup (B + C)$).

The hard point being the closure under * which is a consequence of commutativity. It holds because $(v + V^*)^* = (v + (\{v\} \bigcup V)^*) \bigcup \{0\}$ (the key idea being that $(a(b^*))^* = a^*b^*$ in a commutative monoid). See [Reutenauer 1989] (Proposition 3.5) for details. $\square$

LEMMA 5.13. *The set of non-empty 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 unique label. The set of paths between two given vertices is a regular language; specifically, the set of cycles that begin and end at vertex s is regular. The full set of cycles is the union of those sets over all the (finitely many) vertices, and is consequently also regular. $\square$

Example 5.14. Consider again the VASS of Figure 7 or, rather, their underlying graph. The set of (possibly empty) cycles from a to itself is described by the regular expression $A = ((a_1a_3a_4a_5)|(a_2a_4a_5))^*$ and correspond exactly to the rational language recognised by this expression. Then the set of all (non-empty) cycles in these VASS is the language recognised by the regular expression:

$$(((a_1a_3a_4a_5)|(a_2a_4a_5))A)|(a_3a_4a_5Aa_1)|(a_4a_5A(a_1a_3|a_2))|(a_5A(a_1a_3|a_2)a_4)$$

where each of the four alternatives correspond to the set of (non-empty) cycles from one vertex to itself.

COROLLARY 5.15. *The set of weights of (non-empty) cycles in a VASS is a semi-linear part of $\mathbb{Z}^k$.*

PROOF. Since the weighting function ω is a morphism between $(A, \cdot)$ and $(\mathbb{Z}^k, +)$, it preserves rational parts. Hence, the set of weights of cycles is a rational part of $\mathbb{Z}^k$. Since $+$ is commutative, it is also a semi-linear part. $\square$

Notice that the proofs are constructive. Hence the semi-linear part can be built effectively.

Example 5.16. For concision, we will write here ω_i instead of $\omega(a_i)$.

First, let us look at the weight of cycles from a to itself. By applying the weighting morphism (ω) to A, we obtain the regular expression:

$$((\omega_1 + \omega_3 + \omega_4 + \omega_5)|(\omega_2 + \omega_4 + \omega_5))^*$$

To express this as a semi-linear part, we must change the alternatives (|) into union of sets. This leads to:

$$\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*$$

which is a semi-linear part of $\mathbb{Z}^3$. For the first VASS, this is:

$$\{(0,0,-1),(-1,0,0)\}^* = \{(-k,0,-l)|k,l\in\mathbb{N}\}$$

Next, consider the expression describing cycles from b to itself: $a_3a_4a_5Aa_1$. When applying the weight to it, we obtain:

$$\{\omega_3+\omega_4+\omega_5\}+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*+\{\omega_1\}$$

By commutativity of addition, this can be expressed as the semi-linear part:

$$(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*$$

Again, if we consider the first VASS, this is:

$$(0,0,-1)+\{(0,0,-1),(-1,0,0)\}^* = \{(-k,0,-l)|k,l\in\mathbb{N}, l>0\}$$

Then, we must do the same work for the three other alternatives (corresponding to cycles from a, c and d). This leads, to the following semi-linear parts:

—for a, c and d: $(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^* \bigcup (\omega_2+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*$
—for b: $(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*$

The expression for b is different from the others because non-empty cycles at b must go at least once through the large cycle while other non-empty cycle can go through the small cycle only.

The resulting semi-linear part of $\mathbb{Z}^k$ describing weight of cycles corresponds to the union of these semi-linear parts, namely:

$$\begin{aligned}&\{\ (\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*\ \}\\ \bigcup\quad &\{\ (\omega_2+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*\ \}\end{aligned}$$

For the first VASS, this is:

$$\begin{aligned}((0,0,-1)+\{(0,0,-1),(-1,0,0)\}^*)\bigcup((-1,0,0)+\{(0,0,-1),(-1,0,0)\}^*)\\ =\{(-k,0,-l)|k,l\in\mathbb{N},k+l>0\}\end{aligned}$$

For the second VASS, this is:

$$\begin{aligned}((-1,1,1)+\{(-1,1,1),(3,-1,-2)\}^*)\bigcup((3,-1,-2)+\{(-1,1,1),(3,-1,-2)\}^*)\\ =\{(3l-k,k-l,k-2l)|k,l\in\mathbb{N},k+l>0\}\end{aligned}$$

THEOREM 5.17. *Uniform termination of VASS is in* NPTIME.

PROOF. By Theorem 5.9, a VASS is *not* uniformly terminating if and only if there is a cycle whose weight is in $\mathbb{N}^k$. Since the set of weights of cycles is a semi-linear part of $\mathbb{Z}^k$, it is sufficient to be able to decide whether a linear part of $\mathbb{Z}^k$ intersects $\mathbb{N}^k$ (and try this for each linear part of the union).

Let $U = \{u_1, \cdots, u_p\}$ and $u + U^*$ be a linear part of $\mathbb{Z}^k$. It intersects $\mathbb{N}^k$ if and only if there exist $n_1, \cdots, n_p \in \mathbb{N}$ such that $u + \sum n_i u_i \geq 0$.

This can be solved in NPTIME using usual integer 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 simpler proof can be made for Petri nets. Such a proof can be found in [Moyen 2003], (theorem 60, page 83).

Example 5.18. Consider again the two VASS of Figure 7. The set of weight of non-empty cycles of the first VASS corresponds to the semi-linear part:

$$((0,0,-1) + \{(0,0,-1),(-1,0,0)\}^*) \bigcup ((-1,0,0) + \{(0,0,-1),(-1,0,0)\}^*)$$

The first linear part of the union, $(0,0,-1) + \{(0,0,-1),(-1,0,0)\}^*$, intersect $\mathbb{N}^3$ if and only if there exists $n_1, n_2 \in \mathbb{N}$ such that:

$$(0,0,-1) + n_1 \times (0,0,-1) + n_2 \times (-1,0,0) \geq (0,0,0)$$

This is clearly impossible.

Similarly, the second linear part cannot intersect $\mathbb{N}^3$. Hence, the set of weights of non-empty cycles does not intersect $\mathbb{N}^3$ and the VASS is uniformly terminating.

For the second VASS, the weights correspond to the semi-linear part:

$$((-1,1,1) + \{(-1,1,1),(3,-1,-2)\}^*) \bigcup ((3,-1,-2) + \{(-1,1,1),(3,-1,-2)\}^*)$$

There, for the first linear-part, the system becomes:

$$(-1,1,1) + n_1 \times (-1,1,1) + n_2 \times (3,-1,-2) \geq (0,0,0)$$

Usual Integer Linear Programming techniques show that the system has solution, for example with $n_1 = 1, n_2 = 1$, corresponding to the cycle $(a_1 a_3 a_4 a_5)^2 (a_2 a_4 a_5)$ whose weight is $(1,1,0)$. Hence, the VASS is not uniformly terminating.

However, any infinite walk starting from, for example, the configuration $(a,(0,14,0))$ is not admissible. Deciding whether a given configuration leads to an infinite admissible walk or not is a different problem from uniform termination.

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 requires a specific solution.

5.3 VASS as Resource Control Graphs

Before the formal definition of Resource Control Graphs, we show here how VASS can be used to build proofs of uniform termination of programs.

In the rest of this section, we consider the following size function:

$$||\langle \mathtt{IP}, \sigma \rangle|| = (|\mathbf{stk}_1|, \ldots, |\mathbf{stk}_s|)_{\mathbf{stk}_i \in \mathcal{S}}$$

that is, the vector whose components are the lengths of the different stacks of a given program. Moreover, we use (e_i) to denote the canonical basis of $\mathbb{Z}^k$, that is e_i is the vector whose jth component is $\delta_{i,j}$.

Definition 5.19 *(Weights)*. To each instruction, we assign the following *weight*:

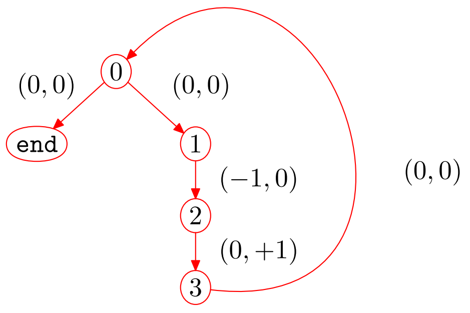

Fig. 8. The Resource Control VASS for the reverse program

—$\omega(\mathbf{r} := \mathtt{pop}(\mathbf{stk}_i)) = -e_i$
—$\omega(\mathtt{push}(\mathbf{r}, \mathbf{stk}_i)) = e_i$
—$\omega(\mathtt{i}) = 0$ for all other instructions.

Definition 5.20 *(Resource Control VASS).* Let p be a program. Its *Resource Control VASS* is a VASS whose underlying graph is the Control Flow Graph of p and edge $\mathtt{i}$ has weight $\omega(\mathtt{i})$ as defined above.

PROPOSITION 5.21. *Let p be a program and G be its Resource Control VASS. If $\theta_0 = \langle IP_0, \sigma_0\rangle \xrightarrow{*} \langle IP_n, \sigma_n\rangle = \theta_n$ is an execution of p, then $(IP_0, ||\theta_0||) \xrightarrow{*} (IP_n, ||\theta_n||)$ is an admissible walk of G with the same trace.*

PROOF. By induction on the length of the execution. Notice that executions leading to errors ($\perp$) are not taken into account here. □

THEOREM 5.22. *Let p be a program and G be its Resource Control VASS. If G is uniformly terminating, then p is uniformly terminating.*

PROOF. If p is not uniformly terminating, then by the previous proposition there exists an infinitely long execution that can be mapped onto an infinite admissible walk. □

Since uniform termination of VASS is decidable, this allows to detect uniform termination of a broad class of programs. Of course, the converse is not true since uniform termination of programs is not decidable.

Example 5.23. The Resource Control VASS of the reverse program is displayed on Figure 8. Since it is uniformly terminating, so is the reverse program.

Weighted graphs, as used in Section 4 to prove Non-Size Increasingness of programs are the special case of VASS when the dimension is one.

6. RESOURCE SYSTEMS WITH STATES

Resource Systems with States (RSS) are a generalisation of the VASS seen in the previous Section. For VASS, the only information kept is a vector of integers, and only addition of vectors can be performed. When modelling programs, this is not sufficient. Indeed, if one wants to closely represent the memory of a stack machine, a vector is not sufficient. Moreover, vector addition is not powerful enough to represent common operations such as copy of a variable ($x := y$).

Hence, we will now relax the constraints on valuations and weights. We will allow valuations to be drawn from any set and allow as weight any function mapping valuations to valuations. Notice that in the case of VASS, each weight is addition of a vector v, which could be represented as the function $\lambda x.x + v$.

For the sake of generality, we will even allow the sets of valuations to be different for each vertex. This may seem 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. See Example 9.3 for more details.

6.1 Graphs and States

Definition 6.1 *(RSS, configurations, walks).* 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 $\omega(a) \in W_{i,j}$ if $s_i \xrightarrow{a} s_j$.

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

A *configuration* is a pair $\eta = (s, v)$ where $s = s_i \in S$ is a vertex of the graph and $v \in V_i$ is a valuation. A configuration is *admissible* if $v \in V_i^+$ is admissible.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (s_n, v_n) \xrightarrow{a_{n+1}} \ldots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \ldots$ and for all $i > 0$, $v_i = \omega(a_i)(v_{i-1})$. A walk is *admissible* if all configurations in it are admissible.

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

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

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 the ring $\mathbb{Z}^k$, 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^+ under 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 whether 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 *(Weight of a path).* Let G be an 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.

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 overload the notation and call it W as well.

In the following, to improve readability, we will write $v \circledast \omega(a)$ instead of $\omega(a)(v)$ and $\omega(a) \fatsemi \omega(b)$ instead of $\omega(b) \circ \omega(a)$. When following paths, we now have: $\omega(ab) = \omega(b) \circ \omega(a) = \omega(a) \fatsemi \omega(b)$. So, this allows for a more natural expression of weights of paths[8].

Example 6.3. For the VASS of the previous Section, we have $V_i = \mathbb{Z}^k$ and $V_i^+ = \mathbb{N}^k$ for all i, and $\omega(a_i) = \lambda x.x + u_i$ for some vector $u_i \in \mathbb{Z}^k$. Or, we could describe VASS by saying that $V = W = \mathbb{Z}^k$, $V^+ = \mathbb{N}^k$ and $\circledast = \fatsemi = +$.

The notation with $\circledast$ and $\fatsemi$ is much more convenient, especially to easily handle weights of paths, as is done in the Lemmas and Theorems of the previous Section.

If we consider V_i as objects 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.

6.2 Properties of RSS

6.2.1 *Order*

Definition 6.4 *(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 $\prec$ to V^+ is a well partial order.

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.

Definition 6.5 *(Monotonicity, positivity)*. Let (G, V, V^+, W, ω) be an ordered RSS. We say that it is *increasing* if all weighting functions $\omega(a_i)$ 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 for each $v \in V^+$ and $v' \in V$, $v \prec v'$ implies $v' \in V^+$.

VASS are both increasing and positive. Monotonicity is the key of Lemma 5.6 while positivity is implicitly used in the proof of Theorem 5.9 to say that the valuation reached after one cycle is still admissible.

Definition 6.6 *(Resource awareness)*. Let G be an ordered RSS and $f : V \to V$ be a function. G is *f-resource aware* if for any walk $(s_0, v_0) \xrightarrow{*} (s_n, v_n)$ we have $v_n \preceq f(v_0)$

6.2.2 *Uniform termination*

Definition 6.7 *(Uniform termination)*. Let G be an RSS. G is *uniformly terminating* if there is no infinite admissible walk over G.

[8]From an algebraic point of view, this means that ω is considered as a morphism between $(A, \cdot)$ and $(W, \fatsemi)$, and $\circledast$ is a right-action of W on V. Moreover, $(W, \fatsemi)$ often appears to be isomorphic to a well known structure (usually a group, such as $(\mathbb{Z}^k, +)$ for VASS).

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

Theorem 5.9 can be generalised to RSS:

THEOREM 6.8. *If G does* not *uniformly terminate, then there is an admissible cycle $(s, v)\xrightarrow{+}(s, u)$ with $v \preceq u$. If G is increasing 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 configurations (s', v_k) with fixed s', 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 an admissible cycle.

If such a 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 not decrease. Positivity is needed to ensure that when going through never decreasing valuations one will not leave V^+. □

PROPOSITION 6.9. *Let $G = (G, V, V^+, W, \omega)$ be an RSS.*

(*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 uniform termination for an ordered RSS G is semi-decidable.*

PROOF.

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

(2) If there are only finitely many valuations, any infinite walk eventually comes back to exactly the same configuration, hence the cycle of Theorem 6.8 becomes $(s, v)\xrightarrow{+}(s, v)$. Then it is possible to compute all the possible weights of cycles (there are only finitely many of them) and check with each valuation whether the condition is met. Notice that this does not require the RSS to be ordered.

(3) By enumerating the cycles and the valuations simultaneously, computing the new valuation after going through the cycle and checking with the ordering whether this satisfies Theorem 6.8.

□

Corollary 5.15 can be generalised:

PROPOSITION 6.10. *If $(W, \circ)$ is commutative, then the set of weights of cycles of an RSS is semi-linear.*

This allows us to easily find candidates for a generalisation of Theorem 5.17 if the set of "positive" weights is easily expressible (as it was the case for VASS). Among other properties: if it is itself semi-linear, then uniform termination is decidable (because intersection between two semi-linear parts is decidable).

6.3 Equational versus constraint based approach

Up to now, the only weights we have considered are functions, meaning that if $s\xrightarrow{a}s'$, for each valuation v there is only one valuation v' such that $(s,v)\xrightarrow{a}(s',v')$. Sometimes, it is more convenient to have several possible results because approximations of the values leads to a loss of information. In this case, the weights considered will be relations rather than functions and we require $v' \in \widehat{\omega}(a)(v)$ rather than $v' = \omega(a)(v)$.

6.3.1 *Constraints RSS*

Definition 6.11 *(Constraints RSS, configurations, walks).*
A *Constraints RSS* is a tuple (G, V, V^+, W, ω) where

—$G = (S, A)$ is a directed graph.

—$V_i^+ \subset V_i$ are, respectively, the sets of *admissible valuations* and *valuations*.

—$W_{i,j} : V_i \to \mathcal{P}(V_j)$ are the sets of *weights*.

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

Configurations and admissible configurations are defined as earlier.

A *walk* is a sequence (finite or not) of configurations $(s_0,v_0)\xrightarrow{a_1}\ldots\xrightarrow{a_n}(s_n,v_n)\xrightarrow{a_{n+1}}\ldots$ such that $s_0\xrightarrow{a_1}s_1\xrightarrow{a_2}\ldots\xrightarrow{a_n}s_n\xrightarrow{a_{n+1}}\ldots$ and for all $i > 0$, $v_i \in \widehat{\omega}(a_i)(v_{i-1})$. A walk is *admissible* if all configurations in it are admissible.

It is important to notice that even if weighting functions return sets (that is, they are relations rather than functions), each walk has to choose one element from this set as a new valuation. That is, we do not consider configurations with sets as valuations, but rather introduce some kind of non-determinism in the RSS. The main use for this will be when some valuations are in no way related to the previous ones and can be anything (*e.g.* if a value is provided via some external mechanism such as a `scanf` instruction).

Definition 6.12 *(Weight of a path).* Let G be an RSS. The weighting function can be canonically extended over all paths in G by choosing $\widehat{\omega}(ab)(x) = \bigcup_{y\in\widehat{\omega}(a)(x)} \widehat{\omega}(b)(y)$.

As earlier, uniform termination means that there exists no infinite admissible walk. However, monotonicity becomes $x \preceq y \Rightarrow \forall x' \in \widehat{\omega}(x), \exists y' \in \widehat{\omega}(y)/x' \preceq y'$.

Then, Theorem 6.8 becomes:

THEOREM 6.13. *Let G be a positive increasing Constraints RSS. G is* not *uniformly terminating if and only if there is an admissible cycle $(s,v_0)\xrightarrow{+}(s,v_1)$ such that $v_0 \preceq v_1$.*

PROOF. If an admissible infinite walk exists, then we can extract from it an admissible cycle in exactly the same way as with Theorem 6.8.

Conversely, if a non-decreasing admissible cycle c exists, let $(s,v_0)\xrightarrow{a}(s',v_0')\xrightarrow{*}(s,v_1)$ be the first, second and last configurations when following the cycle. By hypothesis, $v_0 \preceq v_1$.

Then, there exists $v_1' \in \widehat{\omega}(a)(v_1)$ such that $(s,v_1)\xrightarrow{a}(s',v_1')$ and $v_0' \preceq v_1'$. By positivity of the VASS, v_1' is still admissible.

By iterating this process, we build the admissible cycle $(s,v_1)\xrightarrow{c}(s,v_2)$ with $v_1 \preceq v_2$. Then, this can be done *ad infinitum* thus leading to an admissible infinite walk. □

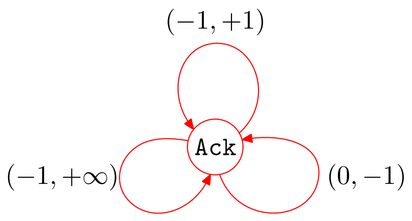

Fig. 9. Constraints VASS for Ackermann's function

6.3.2 *Constraints VASS.*

Let us show how this concept applies to VASS and why it can be useful when studying programs. Remember that $\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$.

Definition 6.14 *(Constraints VASS).* A *Constraints VASS* is a directed graph $G = (S, A)$ together with a *weighting function* $\omega : A \to \overline{\mathbb{Z}}^k$ where k is a fixed integer.

A *configuration* is a pair $\eta = (s, v)$ where $s \in S$ and $v \in \mathbb{Z}^k$. It is *admissible* if $v \in \mathbb{N}^k$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0)\xrightarrow{a_1}\ldots\xrightarrow{a_n}(s_n, v_n)$ such that $s_0\xrightarrow{a_1}s_1\xrightarrow{a_2}\ldots\xrightarrow{a_n}s_n$ and for all $i > 0$, $v_i \le v_{i-1} + \omega(a_i)$. A walk is *admissible* if all configurations in it are admissible.

To express a Constraints VASS as a Constraints RSS, we should consider the weighting function $\widehat{\omega}(a) : \mathbb{Z}^k \to \mathcal{P}(\mathbb{Z}^k)$ such that $\widehat{\omega}(a)(v) = \{v' | v' \le v + \omega(a)\}$. Then, the relation between valuations in a walk will be the general $v_i \in \widehat{\omega}(a_i)(v_{i-1})$. Since, all constraints have the same shape, we can express this in a more readable way. Constraints VASS are positive and increasing. When there is no $+\infty$ in the weights, it is always "best" to choose the greatest possible valuation, that is use the (regular) VASS with the same underlying graph and weighting function.

Example 6.15. Consider the following functional program computing Ackermann's function:

$$\begin{aligned} \mathtt{Ack}(0, n) &\to n + 1 \\ \mathtt{Ack}(m + 1, 0) &\to \mathtt{Ack}(m, 1) \\ \mathtt{Ack}(m + 1, n + 1) &\to \mathtt{Ack}(m, \mathtt{Ack}(m + 1, n)) \end{aligned}$$

For functional programs, an equivalent of the CFG it the *call graph*. There is one vertex for each function symbol (here only one) and one edge for each call (here, 3). Since there are two positive integers in the program, it is natural to choose (m, n) as valuation.

The first line does not perform any call, hence there is no edge corresponding to it in the graph (since termination is studied here, the first line can never lead to non-termination, hence it is safe to have nothing corresponding to it in the graph).

The second line performs one call where the arguments of the function go from $(m + 1, 0)$ to $(m, 1)$, this corresponds to adding $(-1, 1)$ to the valuation.

The third line performs two calls. The inner call is from $\mathtt{Ack}(m+1, n+1)$ to $\mathtt{Ack}(m+1, n)$ (embedded in some context). That is, in this call, the arguments of the function go from $(m + 1, n + 1)$ to $(m + 1, n)$, so the corresponding edge is labelled $(0, -1)$.

However, when considering the outer call in the last line the second argument becomes $\texttt{Ack}(m+1,n)$ which cannot be related to the parameter n in any easy way. So, using a regular VASS, this call would not be representable.

With a Constraints VASS, we can represent this last call. Indeed, not knowing anything on the result simply means that we can relax all constraints on it which will be represented by the vector $(-1,+\infty)$. The constraints VASS for Ackermann's function is displayed on Figure 9.

Since this Constraints VASS is uniformly terminating, so is Ackermann's function.

This both illustrates why Constraints VASS can be useful as well as hints how to apply the ideas behind RCGs to functional programs.

7. RESOURCE CONTROL GRAPHS

Instead of the weighted graphs or VASS used before, we will now use any RSS to model 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 to abstract it. Instead, we consider given a *size function* that associates to each state (or to each store) some size. The size function is unspecified in general. Of course, when using RCG to model 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.

7.1 Resource Control Graphs

Definition 7.1 *(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 $V^+_{\texttt{lbl}}$ be the image by $||\bullet||$ of all states $\langle\texttt{lbl},\sigma\rangle$ for all stores σ.

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

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 valuations V^+ (ordered by $\prec$). $V^+_{\texttt{lbl}}$ being the admissible valuations for vertex $\texttt{lbl}$.

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

LEMMA 7.2. *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.*

THEOREM 7.3. *Let p be a program and G be its RCG. If G is uniformly terminating, then p is also uniformly terminating.*

Example 7.4. A Space-RCG as defined in Section 4 is a special case of general RCG. In this case, $||\theta|| = |\theta|$, this leads to $V^+_{\texttt{lbl}} = V^+ = \mathbb{N}$ for each label $\texttt{lbl}$. Similarly,

$\omega(\mathtt{i}) = \lambda x.x + k_{\mathtt{i}}$ with $k_{\mathtt{i}}$ as in definition 4.1. Since $k \in \mathbb{Z}$, the closure of V^+ by the weighting functions is $V = \mathbb{Z}$.

In this case, resource awareness of the Space-RCG (or β-Space-RCG) guarantees a resource bound on the program execution.

Example 7.5. For a better representation of programs, the size can be the vector where each component is the length of a stack: $||\langle \mathtt{IP}, \sigma \rangle|| = (|\mathbf{stk}_1|, \dots, |\mathbf{stk}_s|)_{\mathbf{stk}_i \in \mathcal{S}}$. This corresponds exactly to what is done with the Resource Control VASS of Section 5.3. As shown, this allows to decide uniform termination of several programs.

This 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 the decreasing in size, but also the increasing. In this way, 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 NPTIME, as we have shown, uniform termination of VASS is in NPTIME. The original SCT, as well as fan-in free δSCT, is PSPACE-complete. However, this simple method does 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 model such kind of programs accurately[9], hence the NP bound is not a big surprise.

Moreover, this method has a fixed definition of size and hence will not 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 integers (represented as letters of the alphabet) will not be analysed correctly.

Example 7.6. However, this representation can be improved. Typically, using Resource Control VASS it is impossible to detect anything happening to registers. If we have a suitable size function $|| \bullet || : \Sigma \to \mathbb{N}$ for registers[10], we can choose $||\langle \mathtt{IP}, \sigma \rangle|| = (||\mathbf{r}_1||, \dots, ||\mathbf{r}_r||)_{\mathbf{r}_i \in \mathcal{R}}$. In this case, depending on the operators, weight could be either vector addition or matrix multiplication (to allow the copy of a register).

Remark 7.7. 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 superset, see the work of Avery [2006].

[9] And cannot even model all those programs due to the restriction on copying variables.

[10] Note that the *size* function used here is in no way related to the *length* of a state. It plays no role when computing the space usage of a state and may also be seen as an ordering over the alphabet.

Remark 7.8. The size function is not specified and may depend on the property one wants 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 registers 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].

Example 7.9. Let us consider the following program, working on integers (that is the alphabet is the set `unsigned` of 32 bits positive integers):

```
0 : $\mathbf{i} := 0$;
1 : if $\mathbf{i} \geq \mathbf{n}$ then goto $5$;
2 : $\mathbf{i} := \mathbf{i} + 1$;
3 : some instructions modifying neither $\mathbf{i}$ nor $\mathbf{n}$
4 : if $\mathbf{i} < \mathbf{n}$ then goto $2$;
5 : $\mathbf{i} := \mathbf{i} + 1$;
end : end;
```

This is simply a loop `for(i=0;i<n;i++)` (in a `C`-like syntax). If we consider a size function that simply takes the vector of the registers, that is $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n})$, 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 $||\langle \mathtt{IP}, \sigma\rangle|| = \mathbf{n} - \mathbf{i}$. 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 this inequality must hold only in the loop. Indeed, at label 5 or after, we may have $\mathbf{i} > \mathbf{n}$. Hence using this size function everywhere would cause troubles since then $||(5, \sigma)||$ will not be admissible.

Having different sets of valuations for each labels, that is a size function operating differently on each label, can solve this problem. By choosing $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n})$ for $\mathtt{IP} = 0, 1, 5, \mathtt{end}$ and $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n}, \mathbf{n} - \mathbf{i})$ otherwise, we can ensure that the "natural" sets of admissible valuations ($\mathbb{N}^2$ and $\mathbb{N}^3$) indeed correspond to the image of the size function (or at least a manageable superset of it).

In this case, of course, we need the weight between labels 1 and 2 to take into account the apparition of a new component in the valuation. Here, this can be done using a matrix multiplication since the new component in the valuation is a linear combination of the existing ones. See Example 9.3 for the complete construction of the RCG.

7.2 Constraints RCG

Constraints RSS can also be used instead of RSS to model programs and build RCG as was done with the Ackermann's function of Example 6.15. In that case, the relation required between weights and sizes is:

for all states θ verifying $p \vdash \theta \xrightarrow{\mathtt{i}} \theta'$, $||\theta'|| \in \widehat{\omega}(\mathtt{i})(||\theta||)$.

Then, the simulation Lemma and uniform termination Theorem are still true:

LEMMA 7.10. *Let p be a program, G be its Constraints 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.*

PROOF. Because $||\theta_i||$ belongs to $\widehat{\omega}(a)(\theta_{i-1})$ and can thus always be chosen as the new valuation. □

THEOREM 7.11. *Let p be a program and G be its RCG. If G is uniformly terminating, then p is also uniformly terminating.*

8. δ-SIZE CHANGE TERMINATION

In Section 4, we have used RCG in order to have an analysis of running space similar to the Non Size Increasing approach of Hofmann. Here, we will use RCG to analyse termination of programs in a way similar to the Size Change Termination of Lee, Jones and Ben-Amram and, more precisely, to the δ-Size Change Termination of Ben-Amram.

We consider here the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring and denotes $\min$ as $\oplus$ and $+$ as $\otimes$. These operations are canonically extended to define multiplication of matrices[11] from $\mathcal{M}(\overline{\mathbb{Z}})$.

8.1 Matrices and graphs

Definition 8.1 *(Constraint graph).* Let M be a square matrix of dimension n. Its *constraint graph* is a weighted directed graph G such that:

—There are n vertices $X_i, 1 \leq i \leq n$ plus an extra vertex Y.
—If $M_{i,j} \neq +\infty$, there is an edge of weight $M_{i,j}$ between X_i and X_j.
—There is an edge of weight 0 between Y and X_i, for all i.

Definition 8.2 *(l-weight).* Let G be a directed weighted graph. The *l-weight between a and b* is the minimum weight of all paths of length l between a and b and $+\infty$ if there is no such path.

The coefficient $M^k_{i,j}$ is the k-weight between X_i and X_j in the constraint graph of M.

LEMMA 8.3. *The system $X \leq X \otimes M$ has a solution if and only if there is no strictly negative coefficient in the diagonal of M^k, for all k. In that case, it admits a non-negative solution.*
It is possible to decide in polynomial time whether such a system admits a solution.

PROOF. The matrix inequality corresponds to the set of inequalities $\{X_j \leq \min_i(X_i + M_{i,j})\}$ which can, without modifying the set of solutions, be expressed as $\{X_j \leq X_i + M_{i,j}\}$.
If there is no strictly negative coefficient in the diagonal of M^k, that means that the constraints graph G has no cycle of strictly negative weight. In this case, we can choose for X_i the value of the shortest path to reach it from Y. This is well defined because there is no cycle of strictly negative weight and provides a solution for the system because $X_j \leq X_i + M_{i,j}$ by definition of shortest paths.
Conversely, if there is a path of strictly negative weight, then it is easy to see that by adding the inequations corresponding to the edges in this path one will eventually reach an inequation $X_i < X_i$ and the system has no solution.
If there is a solution, then $X + (1, \dots, 1)$ is also a solution. Hence, there exists a solution where all values are positive.

[11] That is, given two matrices A and B, $(A \oplus B)_{i,j} = A_{i,j} \oplus B_{i,j} = \min(A_{i,j}, B_{i,j})$ and $(A \otimes B)_{i,j} = \bigoplus_k A_{i,k} \otimes B_{k,j} = \min_k(A_{i,k} + B_{k,j})$. Similarly, if X is a vector and M a matrix, then $(X \otimes M)_j = \bigoplus_k V_k \otimes M_{k,j} = \min_k(V_k + M_{k,j})$.

The system admits a solution if and only if the constraint graph has no cycle of strictly negative weight. This can be decided in polynomial time by Bellman-Ford's algorithm. □

8.2 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) of Ben-Amram [2006]. Here, we use matrices rather than Size Change Graphs thus following the work 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).

In this whole section, we consider a fixed program p, and for each label $\texttt{lbl}_a$ in it 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 we consider given a size function $|| \bullet ||$ such that for each label $\texttt{lbl}_a$ and for each store σ, $||\langle \texttt{lbl}_a, \sigma\rangle|| \in V_a^+$.

Definition 8.4 *(Size Change Matrix)*. Let $\texttt{i}$ be an instruction in p corresponding to an edge between $\texttt{lbl}_a$ and $\texttt{lbl}_b$ in G. The *Size Change Matrix* (SCT matrix) of $\texttt{i}$ is a matrix $M^{(\texttt{i})}$ of $\mathcal{M}_{k_a,k_b}(\overline{\mathbb{Z}})$ such that for all states θ_a with $p \vdash \theta_a \xrightarrow{\texttt{i}} \theta_b$, $||\theta_b|| \leq ||\theta_a|| \otimes M^{(\texttt{i})}$.

This means that if $||\theta_a|| = (x_1, \cdots, x_{k_a})$ and $||\theta_b|| = (y_1, \cdots, y_{k_b})$, we have for each j: $y_j \leq \min_k\{x_k + M^{(\texttt{i})}_{k,j}\}$ where the coefficients of $M^{(\texttt{i})}$ can be any integer or $+\infty$.

Definition 8.5 *(Size Change RCG)*. The *Size Change RCG* (SCT-RCG) of p is the Constraints RCG for p build with admissible valuations $\mathbb{N}^{k_a}$, and valuations $\mathbb{Z}^{k_a}$ for vertex $\texttt{lbl}_a$. The weight for edge $\texttt{i}$ is such that $\widehat{\omega}(\texttt{i})(v) = \{v' | v' \leq v \otimes M^{(\texttt{i})}\}$ where $M^{(\texttt{i})}$ is the SCT matrix for $\texttt{i}$.

As for Constraints VASS, the common shape of constraints allows to use a weighting function $\omega(\texttt{i}) = M^{(\texttt{i})}$ instead of the weighting relation $\widehat{\omega}$ and ask along a walk that $v_i \leq \omega(a_i)(v_{i-1})$ rather than $v_i \in \widehat{\omega}(a_i)(v_{i-1})$.

The uniform termination Theorem for Constraints RCG (Theorem 7.11) tells us that if the SCT-RCG is uniformly terminating then so is p.

SCT-RCG are both increasing and positive, so it will be possible to apply Theorem 6.13.

THEOREM 8.6. *Let G be the SCT-RCG of p. It is uniformly terminating if and only if it does not contains a cycle c of weight $M^{(c)}$ such that $X \leq X \otimes M^{(c)}$ admits a solution.*

PROOF. If the system $X \leq X \otimes M^{(c)}$ admits a solution, then it admits an arbitrarily large solution. Hence, there exists an admissible cycle $(s, X) \xrightarrow{c} (s, X \otimes M^{(c)})$ and by Theorem 6.13, the SCT-RCG is not uniformly terminating.

Conversely, if the SCT-RCG is not uniformely terminating then, by Theorem 6.13 there exists a cycle of weight M such that $X \leq X \otimes M$ has a solution. □

Remark 8.7. The readers familiar with the original works of Lee et al. [2001] or Ben-Amram [2006] may wonder why there is no idempotence condition in Theorem 8.6. As a matter of fact, it happens that any square matrix M on the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring has a power $M' = M^k$ which is *strongly sign idempotent*, that is the coefficients $M'^n_{i,j}$, for all $n > 0$ all have the same sign.

The matrices we use here, as well as the Size Change Graphs in the other works, represent the flow of data. The idea behind idempotence is that we want to detect a cycle in the program such that the corresponding flow of data is also circular, that is each variable flows to itself.

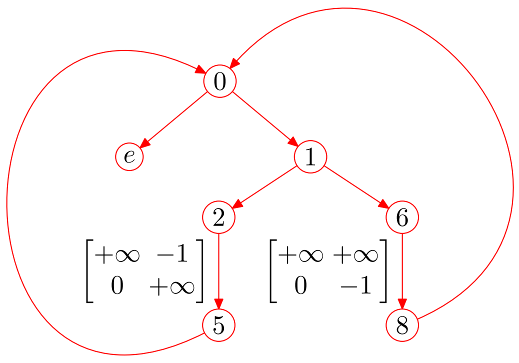
0
e
1
2
6
5
8
$\begin{bmatrix} +\infty & -1 \\ 0 & +\infty \end{bmatrix}$
$\begin{bmatrix} +\infty & +\infty \\ 0 & -1 \end{bmatrix}$

Fig. 10. A Size Change Termination RCG.

The dangerous cycles (with respect to termination) are those who (i) have an idempotent flow of data and (ii) do not have a decrease in one of the data. Indeed, these cycles could be repeated infinitely many time, leading to an infinite execution.

However, as stated for matrices at the beginning of the Remark, each flow of data eventually becomes idempotent if repeated several times. Hence, finding a cycle whose weight M is such that $X \leq X \otimes M$ admits a solution is sufficient to get a cycle with idempotent flow of data by repeating this cycle.

With RCG, the notion of valuation makes the inequality $X \leq X \otimes M$ pretty natural, since it exactly correspond to what happen to valuations after going through the cycle. The original RCG works, however, did not have this notion of valuation but only the matrices (or graphs), seen as a description of the modification on the size of variables (independently to the actual value of variable, that is the valuations). Hence, the idempotence condition was natural in this framework but the notion of RCG shows that one can actually get rid of it.

Notice that the flow of data is somewhat taken into account in Lemma 8.3 where we consider the sign of the coefficients of the diagonal of M^k. The coefficients on the diagonal of M^k explain how the data flows from x_i to itself after repeating the cycle k times.

Notice that by Lemma 8.3, the individual condition on cycles is decidable in polynomial time. The general condition, however, is undecidable. Nevertheless, if the matrices are *fan-in free*, that is in each column 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 the SCT-RCG 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).

Example 8.8. Consider the following program (adapted from [Lee et al. 2001] fifth example):

```
0 : if y = 0 then goto end;        5 : goto0;
1 : if x = 0 then goto 6;          6 : x := y;
2 : a := x;                        7 : y := y − 1;
3 : x := y;                        8 : goto0;
4 : y := a − 1;                  end : end;
```

It can be proved terminating by choosing the size function $||\theta|| = (\mathbf{x}, \mathbf{y}, \mathbf{a})$. With this size, its SCT-RCG is displayed on Figure 10. For convenience reasons, instructions $2-4$, as well as $6-7$ have been represented as a single edge (with a single matrix). This allows 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 the SCT-RCG is uniformly terminating, so is the program.

When working with this simple Size Change Principle (or any other restriction where there can be only finitely many different weights), Theorem 8.6 gives an algorithmic way of detecting uniform termination of the SCT-RCG. Indeed, there are only finitely many different weights, hence there are only finitely many tuples (s, M, r) such that there exists a path from s to r whose weight is M. Then, it is possible to build all these tuples in an incremental way by starting with tuples (s, M, r) corresponding to each edge of the SCT-RCG and add new tuples by composing existing ones with matching edges. This is the core idea of the "graph algorithm" of Lee et al. [2001].

9. MORE ON MATRICES

9.1 Matrices Multiplication System with States

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 9.1 *(Matrices Multiplication System with States).* A *Matrices Multiplication System with States* (MMSS) is an 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.

—$\mathbin{⨾} = \circledast = \times$.

Using this, it is quite easy to model copy instructions of counters machines ($\mathbf{x} := \mathbf{y}$) 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 known as *homogeneous coordinates*[12]. Simply represent the n counters as a $n+1$ components vector whose first component is always 1. Then, increment or decrement of a variable just becomes a linear combination of components of the vector which can

[12]Homogeneous coordinates were originally introduced by A. F. Möbius. They are used, among other, in computer graphics for exactly the same purposes as we do here, that is representing a translation by means of matrix multiplication.

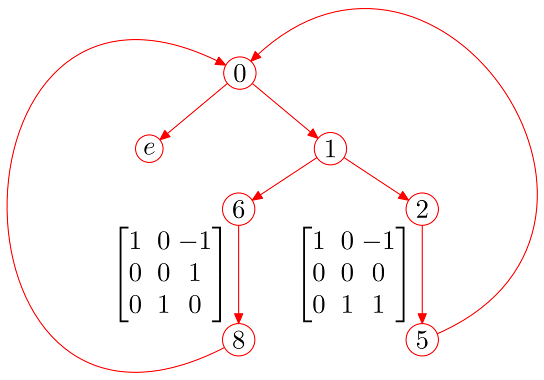
0
e
1
6
2
8
5
$\begin{bmatrix}1&0&-1\\0&0&1\\0&1&0\end{bmatrix}$
$\begin{bmatrix}1&0&-1\\0&0&0\\0&1&1\end{bmatrix}$

Fig. 11. MMSS as a RCG.

perfectly be done with matrices multiplication. For example, here is how one can model the copy ($\mathbf{x} := \mathbf{y}$) and the increment ($\mathbf{x} := \mathbf{x} + 1$).

$$(1,x,y) \times \begin{bmatrix}1&0&0\\0&0&0\\0&1&1\end{bmatrix} = (1,y,y) \qquad (1,x,y) \times \begin{bmatrix}1&1&0\\0&1&0\\0&0&1\end{bmatrix} = (1,x+1,y)$$

Example 9.2. Using homogeneous coordinates, the program of Example 8.8 has the MMSS depicted on Figure 11. Here, matrices multiplication is done on the usual $(\mathbb{Z}, +, \times)$ ring and not on the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring as for SCT-RCG.

Example 9.3. Similarly, use of homogeneous coordinates allows to build a MMSS to prove uniform termination of the program of Example 7.9. It is depicted on the left part of Figure 12 (where label 3 has been omitted). The interesting thing here is the use of vectors of different lengths at different labels, thus allowing to add the constraint $\mathbf{n} - \mathbf{i} \geq 0$ only inside the loop. This example shows both the use of disjoint sets of valuations and how to work with the functional inequalities of Avery [2006].

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 it immediately after. The net effect is null but if x is 0, the intermediate valuation is not admissible. This can still be done 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 9.4. *Uniform termination of MMSS is not decidable.*

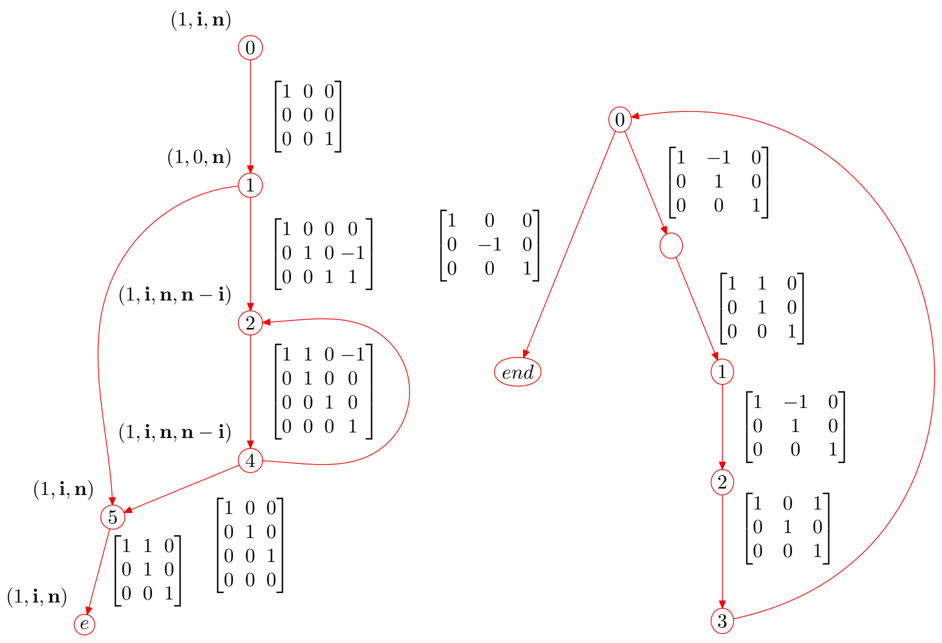

Fig. 12. MMSS for loop and unary addition.

Example 9.5. Consider the following program, performing addition in unary (that is, repeatedly decrementing **x** and incrementing **y** until **x** is 0).

```
  0 : if x = 0 then goto end;   |    3 : goto 0;
  1 : x := x − 1;               |  end : end;
  2 : y := y + 1;               |
```

Right side of Figure 12 depicts a MMSS for this program such that there is a one-to-one correspondence between executions of the program and admissible walks of the MMSS. The size used is $(1, \mathbf{x}, \mathbf{y})$, the 1 being here because of homogeneous coordinates. Notice that we need to add an intermediate label for the $\mathbf{x} \neq 0$ branch of the test in order to generate the temporary valuation containing $\mathbf{x} - 1$, only used to force admissible walks with $\mathbf{x} = 0$ to take the other branch.

On the other branch of the test, the -1 in the center of the matrice ensures that if $x > 0$ in the valuation at vertex 0, then following this edge will lead to a non-admissible valuation. That is, this edge can only be followed if $x = 0$.

Since such a construction can be done for any counter machine (the unary addition program uses all possible instructions for counter machines) and since counter machines are Turing-complete, this shows why uniform termination of MMSS is not decidable in general.

This simulation of programs by matrices multiplications rises a surprising question. Indeed, matrices multiplications are only able to perform linear operations on data. While obviously some programs can perform non-linear operations.

This apparent contradiction is solved when we think more closely on how RSS work. Each walk in a MMSS corresponds to a matrix multiplication (because ω is a morphism), hence to a linear transformation on data. However, two different walks give rise to two different matrices, hence two different linear transformations.

When simulating a program, each different data will go through a different (admissible) walk in the MMSS. Hence, each different value will pass through a different linear transformation. Of course, the other walks (that is, the other linear transformations) also exist and are considered on this data when looking at the set of walks, but non-admissibility allows to dismiss them and only keep one.

So, from a transformation point of view, we can look at MMSS as a set of linear transformations and the admissibility mechanism selects the proper transformation to apply on each piece of data.

For example, if we consider a program performing multiplication of two integers x and y, it will likely be a loop on x, adding y to the result each time. The corresponding MMSS will have several paths (infinitely many) that can each be candidate for a walk once actual data is provided. Different paths correspond to following the loop $1, 2, 3, \ldots, k, \ldots$ times. Then, the walk corresponding to each of these paths will perform the linear transformation $(1, x, y) \mapsto (1, x - k, ky)$ representable by the matrix:

$$\begin{bmatrix} 1 & -k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & k \end{bmatrix}$$

However, when performing all these transformations on actual data, only those with $k \leq x$ have an admissible result and only the one with $k = x$ has all its intermediate valuations admissible. So, the admissibility mechanism selects the right linear transformation to apply.

That means that when simulating a program computing a (non-linear) function by a MMSS, the simulation actually consider the function as being *piecewise linear*, computes the result of all the possible linear transformations implied and selects the one corresponding to current data. In general, it is possible that each linear transformation is only valid for a single value.

9.2 Tensors

Moreover, 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 may 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 could lead to a tensor algebra representing high order programs.

Example 9.6. Here is a tensor representing the MMSS of the unary addition (as depicted in Figure 12). This is simply the connectivity matrix of the graph where each edge is itself weighted by a matrix.

$$\begin{bmatrix}
0 & \begin{bmatrix}1 & -1 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} & 0 & 0 & 0 & \begin{bmatrix}1 & 0 & 0\\0 & -1 & 0\\0 & 0 & 1\end{bmatrix}\\
0 & 0 & \begin{bmatrix}1 & 1 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} & 0 & 0 & 0\\
0 & 0 & 0 & \begin{bmatrix}1 & -1 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} & 0 & 0\\
0 & 0 & 0 & 0 & \begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} & 0\\
\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

9.3 Polynomial time

Another interesting approach of program analysis using matrices is the one done by Niggl and Wunderlich [2006] and Kristiansen and Jones [2005]. The programs they study are similar to our stack 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.

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. Certificates for loops are some kind of multiplicative closure of the certificate for the body and certificate for `if` statements are the least upper bound 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 weights of instructions 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.

10. 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. Analyses 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 Graphs, thus showing the adaptability of the tool.

Moreover, other analyses 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.

Acknowledgements

Many thanks to A. Ben-Amram for pointing out critical flaws in an earlier version of the proof of Theorem 5.17. Thanks also to M. Hofmann for pointing out the name "homogeneous coordinates" and its use in computer graphics.

Many thanks also to the anonymous referees whose lengthy comments helped to greatly improve the presentation of the results.

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, 265–279.

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*, M. Hagiya and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 3945. Springer.

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. *ACM Transactions on Programming Languages and Systems*. To appear.

BONFANTE, G., MARION, J.-Y., AND MOYEN, J.-Y. 2007. Quasi-interpretation: a way to control resources. *Theoretical Computer Science*. To appear, accessible `http://www.loria.fr/~marionjy/Research/Publications/Articles/TCS.pdf`.

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*, 98–117. 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.

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

JONES, N. D. 1997. *Computability and Complexity, from a Programming Perspective*. MIT press.

KRISTIANSEN, L. AND JONES, N. D. 2005. The flow of data and the complexity of algorithms. In *CiE'05:New Computational Paradigms*, Cooper, Lwe, and Torenvliet, Eds. Lecture Notes in Computer Science, vol. 3526. Springer, 263–274.

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.

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

NIELSON, F., NIELSON, H. R., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer.

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.

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

Received September 2006; revised November 2007; accepted hopefully someday