Heap-size analysis for assembly programs

Jean-Yves Marion, Jean-Yves Moyen

Heap-size analysis for assembly programs

Jean-Yves Marion*
INPL-Loria – École Nationale supérieure des Mines de Nancy
Campus scientifique BP239
F-54506 Vandœuvre-lès-Nancy Cédex
Jean-Yves.Marion@loria.fr

Jean-Yves Moyen†
LIPN-CNRS – Université Paris 13
99 avenue J.-B. Clément
F-93430 Villetaneuse
jean-Yves.Moyen@lipn.univ-paris13.fr

ABSTRACT

Our objective is to propose methods for resource-aware compilation inspired by the implicit complexity community. We consider a small assembly-like language and we build abstract finite machine models in order to predict a bound on the maximal heap usage of a program. We propose a polynomial time procedure to detect and certify a broad and meaningful class of non-size increasing programs, which run in a constant size heap. We end by discussing about programs running in linear heap space and discussing how to capture logarithmic space computation.

1. INTRODUCTION

1.1 Motivations

The goal of this study is an attempt to predict and control computational resources like space or time, which are used during the execution of a bytecode program. We focus here on heap space management. For this, we have chosen a bytecode language, which is close to a fragment of C. The machine architecture contains registers and a heap memory. The set of instructions consists in arithmetic operations, conditionals and jumps. The memory management is performed by two low level operations *new* and *free* dealing with dynamic memory allocations.

We present a data flow analysis of the low-level language sketched by means of "Resource Control Graph", and we think that this is a generic concept from which several memory management policies could be checked. We establish a criterion which implies that each execution of a program needs only a constant number of dynamic memory allocations. Such programs were dubbed *Non Size Increasing* (NSI) by Hofmann, who has studied them in the context of functional programming with a linear type discipline [13, 14]. Here, we propose a criterion which is decidable in polynomial time. Besides the interest of this criterion to detect NSI programs, it also illustrates the possibilities and advantages of the concept of Resource Control Graph. Lastly, we describe three other applications: (i) computations with a linear amount of space, (ii) LOGSPACE computations, and (iii) termination by lack of memory.

The goal is to develop an efficient static analysis in order to determine whether or not a system is *heap safe*. Such resource analysis allows to determine how much memory should be allocated and so prevents memory overflow. This is of course crucial for many critical applications, and has strong impact in computer security.

There are several approaches which are trying to solve this problem. The first protection mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could crash unpredictably by memory leak. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a lower bound on the memory while static analysis gives an upper bound. The gap between both bounds is of some value in practical. Lastly, the third approach is type checking done by a bytecode verifier. In an untrusted environment (like embedded systems), the type protection policy (Java or .Net) does not allow dynamic allocation. Actually, the former approach relies on a high-level language, which captures and deals with memory allocation features [5]. 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.

There are other motivations, which are more theoretical. Indeed a lot of works have been done the last ten years to provide syntactic characterisations of complexity classes, see [6, 20] among others. Those characterisations are the bare bone of recent research on delineating broad classes of programs that run in some amount of time or space, like Hofmann, but also Niggl and Wunderlich [26], Amadio, Coupet-Grimal, Dal Zilio and Jakubiec [4], and Bonfante, Marion, Moyen [7].

Lastly, the method is a first step for us to prove imperative program termination by adapting ideas of Lee, Jones and Ben-Amram [17] or Abel and Altenkirch [1]. The intuition is that a program terminates when there is no more resource to consume.

1.2 Coping with undecidability

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

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

The study of extensional complexity quickly reaches the boundary of Rice's theorem. Any extensional property of programs is

*Partly supported by the CRISS project.

†Partly supported by the CRISS and NoCost projects.

either trivial or undecidable. Intuition and empirical results points out that intensional properties are even harder to decide. Section 2 will formalise this impression.

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

The property that we study here is Non Size Increasingness (NSI) as first studied by Hofmann [13]. A program is NSI if its computation does not require more memory than what is initially allocated (*i.e.* the total memory usage is the size of the input, plus some constant amount). This is an intensional property and this is an undecidable property.

When studying intensional properties, two different kind of approaches exist. The first one consist of restricting the syntax of programs so that any program written necessarily has the wanted property. This is in the line of the works on primitive recursive functions where the recurrence schemata is restricted to only primitive recursion. This approach gives many satisfactory results, such as the characterisations of PTIME by Cobham [10] or Bellantoni and Cook [6], the works of Leivant and Marion on tiering and predicative analysis [20] or the works of Jones on CONS-free programs [15]. On the logical side, this leads to explicit management of resources in Linear Logic [12].

All these characterisations usually have the very nice property of *extensional completeness* in the sense that, *e.g.* , every polynomial time computable function can be computed by a bounded recursive function (Cobham). Unfortunately, they're usually very poor on the *intensional completeness*, meaning that very few programs fit in the characterisation [11] and programmers have to rewrite their programs in a non-natural way.

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

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

A more serious example of this kind of characterisation is the Size Change Principle [17]. The set FLOW$^\omega$ that is build during the analysis contains all "well-formed call sequences". Every execution of the program can be mapped to a well-formed call sequence but several (most) of the call sequences do not correspond to any execution of the program. Then, properties (termination) of call sequences in FLOW$^\omega$ are necessarily shared by all execution of the program.

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

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.

Now, what is the situation on the Non Size Increasingness side? All the existing works, either by Hofmann [13] or Aehlig and Schwichtenberg [2] or Aspinall and Compagnoni [5] belongs to the first family of methods. In each case, resource management has to be done by the programmer via the diamond type, meaning that one has to explicitly reuse pointers that are no more needed (via the linear typing).

This work is intended to belong to the second family of methods. This means that programs do not have to reuse pointers but the analysis will discover if a reuse is possible. This has a nice consequence in the way that programmers can now freely allocate and de-allocate memory (with the diamonds, one cannot allocate and everything de-allocated is gone for good). So our analysis do not prevent `malloc`.

Of course, in order to have decidability back, we have to loosen semantics as mentioned above. In our case, similarly to the Size Change Principle, this is done by "rendering tests non deterministic", that is considering that any branch of a `if` can be taken without looking at the actual result of the test.

And of course, there are several programs that are Non Size Increasing but won't be analysed correctly. The set of falsely rejected programs may appear big at first, however, it is not bigger than in Hofmann's framework since we can capture any algorithm that is writable in this model. We will discuss this further at the end of Section 5.2.

2. SOME DECIDABILITY ISSUES

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

Rice's theorem [28] implies that any *extensional* property of Turing Machines (or programs) is either trivial or undecidable. An extensional property is one that depends only of the inputs and the output of the machines, *i.e.* that depends only of the function computed by the machine. A direct application of this theorem leads to the following result:

PROPOSITION 1. *The set of programs p whose output size is bounded by their input size ($|p(x)| \leq |x| + c^{te}$) is not recursive.*

However, the Non Size Increasingness property that we want to study is stronger. Indeed, there exists programs who need tremendous amount of memory to compute but only output a small result. Typically, any program solving a decision problem will only output either "yes" or "no", *i.e.* bounded size information, but may require arbitrary large amount of memory to compute this answer.

The NSI property that we study is actually an *intensional* property of programs. It depends not only on the inputs and outputs but also on the way the computation is performed. Intensional properties are really properties of programs and not properties of functions. Indeed, a given function can be computed by several programs (*e.g.* , Fibonacci's numbers can be computed with a naive exponential time algorithm or by a dynamic programming, linear time algorithm). Extensional properties depending only on the inputs/outputs of the programs are shared by all programs computing the same function. Intensional properties are not. For example, the intensional property of "computing in polynomial time" is not shared by all the program computing Fibonacci's numbers.

Intuitively, intensional properties seem even harder to decide that

extensional ones. This can be formalised a bit by the following theorem.

Here, we consider that a machine is NSI if and only if it never require more space than a constant value plus the initial space (needed to store the input).

THEOREM 1. *Let p be a program. The question "is p NSI?" is undecidable* even if we know that p uniformly terminates.

Here, uniform termination means that the program terminates for all inputs.

PROOF. Let q be a program and consider the program p that works as follows:

- p answer 1 if its input is 0.
- On input $x \neq 0$, p simulates $q(0)$ for x steps.
  - If $q(0)$ halts within x steps, then p answers 0.
  - Else, p answers 2^x (or any other large value depending on x).

Obviously, p uniformly terminates. Is it NSI?

If $q(0)$ terminates, then it does so in n steps. The space used to compute p is bounded by $2^n + S_q(0)$, where $S_q(0)$ is the space needed to compute $q(0)$, *i.e.* a constant value and so p is NSI. If $q(0)$ does not terminates, then p compute the exponential and is certainly not NSI.

So, p is NSI if and only if $q(0)$ terminates. Since the halting problem is not decidable, so is belonging to NSI. □

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

Notice that this proof can be easily adapted to show the undecidability of any complexity class (of programs). It is sufficient to change the function computed by p if $q(0)$ does not terminates.

Remark 1. A similar proof, for the undecidability of running in polynomial time, based on Hilbert's tenth problem[23] ($p(x+1) = 0$ if some polynomial P has a root x, $2 \times p(x)$ otherwise) was presented as the Geocal ICC workshop in Feb. 2006 by Terui. This work has been done independently from a similar result presented by Marion in March 2000 at a seminar in ENS Lyon. This is kind of a folklore result but is nonetheless worth mentioning because lot of confusion is done on the subject.

The above result can be improved. Indeed, the set of programs that run in polynomial time, or which are NSI, is Σ_2-complete. Recall, that a typical Σ_2-complete set is the set of partial computable functions.

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

Next, let $[\![p]\!]$ be the function computed by the program p with respect to an acceptable enumeration of programs. We refer to Rogers' textbook [16] for background. Say that $T_p(x)$ is the number of steps to execute the program p on input x with respect to some universal (Turing) machine.

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

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

THEOREM 2. *The set A_T is Σ_2-complete.*

PROOF. It is clear that the statement which defines A_T is a Σ_2 statement.

Let B be any Σ_2 set defined as follows:

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

We prove that B is reducible to A_T.

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

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

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

□

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

COROLLARY 1. *Let*

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

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

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

- Non-Size Increasing programs (for linear functions and A_S).
- PTIME (for polynomials functions and A_T).
- LOGSPACE, PSPACE, ... any classical complexity class.

3. PROGRAMS

3.1 Syntax

Definition 1. A program is defined by the following grammar:

(Programs)			$p ::=$	$\mathtt{lbl}_1 : i_1; \ldots \mathtt{lbl}_n : i_n;$
(Instructions)	$\mathcal{I}$	$\ni$	$i ::=$	$\mathbf{r}_1, \cdots, \mathbf{r}_l := \mathtt{op}(\mathbf{r}'_1, \cdots, \mathbf{r}'_k) \mid$ $\mathbf{r} := *\mathbf{r}' \mid *\mathbf{r} := \mathbf{r}' \mid$ $\mathbf{r} := \mathtt{new}\, n \mid \mathtt{free}\, \mathbf{r} \mid$ $\mathtt{jmp}\ \mathtt{lbl} \mid \mathtt{jz}\ \mathbf{r}\ \mathtt{lbl}_0\ \mathtt{lbl}_1 \mid$ $\mathtt{end}$
(Labels)	$\mathcal{L}$	$\ni$	$\mathtt{lbl}$	finite set of labels
(Registers)	$\mathcal{R}$	$\ni$	$\mathbf{r}$	finite set of registers
(Operators)	$\mathcal{O}$	$\ni$	$\mathtt{op}$	finite set of operators

Each operator has a fixed arity k and co-arity l 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 $\mathtt{instr} : \mathcal{L} \to \mathcal{I}$ such that $\mathtt{instr}(\mathtt{lbl}_k) = i_k$.

Machines that we consider are close to the RAM-Machines. There is a memory device, similar to heaps on computers, and a finite number of registers. The heap consists in an unbounded number of memory cells. Each cell stores an integer. Each register also stores an integer whose value may be the address of a heap-cell.

Roughly speaking, $*\mathbf{r}$ denotes the value of the memory cell whose address is stored in $\mathbf{r}$. The instruction `jmp lbl` gives the control to label `lbl`, `jz` $\mathbf{r}$ `lbl lbl'` gives the control to either `lbl` (if $\mathbf{r}$ contains 0) or `lbl'`. The instruction `new` n allocates n consecutive cells of memory in the heap and returns the address of the first one and `free` frees the memory cell in the heap at the given address.

Memory allocation is done by block because we may require two given to cells have consecutive addresses (*e.g.* to represent lists) but freeing memory is done cell by cell (unlike, *e.g.* , in `C`) to avoid remembering the size of each allocated block.

Example 1. The following program reverses a list. A list is here represented by a set of positions, each position consists in two consecutive memory cells. The first cell contains the value of the element in the list and the second contains the address of the next position (or 0 for the end of the list). For the sake of clarity, only labels which play a special role (*e.g.* destination of jumps) have been mentioned in this example.

```
begin : next := 0; jz r0 end loop;
 loop : val := *r0; tmp := r0 + 1; free r0;
        r0 := *tmp; free tmp;
        r1 := new 2; *r1 := val; r1 + +; *r1 := next; r1 − −;
        next := r1; jz r0 end loop;
  end : end;
```

Of course, the use of low-level data structure may imply tedious pointer management. However, we could introduce new operators acting both on registers and on the heap that can be seen as macros. For lists, we'll need a `cons` operator taking a value and a list (*i.e.* , an address in the heap) and returning a list (after allocating 2 cells on the heap) and a `destr` operator taking a list and returning a value (the head) and a list (the tail) after de-allocating two cells that can be formally defined as:

- $\mathbf{l}$:= `cons`($\mathbf{a}$, $\mathbf{l'}$) is equivalent to $\mathbf{l}$:= `new` 2; $*\mathbf{l}$:= $\mathbf{a}$; $\mathbf{l}$ + +; $*\mathbf{l}$:= $\mathbf{l'}$; $\mathbf{l}$ − −;
- $\mathbf{a}, \mathbf{l'}$:= `destr`($\mathbf{l}$) is equivalent to $\mathbf{a}$:= $*\mathbf{l}$; **tmp** := $\mathbf{l}$ + 1; `free` $\mathbf{l}$; $\mathbf{l'}$:= $*$**tmp**; `free` **tmp**;

Using these, the program thus becomes:

```
begin :next := 0; lbl1 : jz r0 end loop;
 loop :val, r0 := destr(r0);
 lbl2 :r1 := cons(val, next);
 lbl3 :next := r1; lbl4 : jz r0 end loop;
  end :end;
```

We illustrate an execution of the program in Figure 1.

Notice that if one wants to write the reverse program within Hofmann's or Aspinall and Compagnoni's formalism, one cannot use `new` and `free` and has to explicitly keeps the values of pointer (*i.e.* the diamonds) and tells when they have to be reused.

3.2 Semantics

The domain of the computation is $\mathcal{V} = \mathbb{N}$, and we interpret each operator `op` by a function $[\![\texttt{op}]\!]$ of the same arity an co-arity over $\mathcal{V}$.

An *environment* is a mapping $\sigma : \mathcal{R} \to \mathcal{V}$, which associates to each register a value. We note $\{\mathbf{r}_i \leftarrow v\}\sigma$ the environment that maps $\mathbf{r}_i$ to v and $\mathbf{r}_j$ to $\sigma(\mathbf{r}_j)$ $(j \neq i)$. When no ambiguity can arise, we just write $\mathbf{r}$ instead of $\sigma(\mathbf{r})$.

A *heap* is a mapping $\mu : \mathbb{N} \to \mathcal{V} \bigcup \{\bot\}$, which returns a value if a heap address is allocated. Otherwise, it returns $\bot$ when a heap address is unallocated. We note $\{n \leftarrow v\}\mu$ to say that the heap is updated in such a way that $\{n \leftarrow v\}\mu(n) = v$ and $\{n \leftarrow v\}\mu(m) = \mu(m)$ $(m \neq n)$.

Memory allocation is performed by some external mechanism similar to a system call. In our framework, we have a predicate `memfree`(k, n) which tests whether the memory cells between k and $k + n - 1$ are currently unallocated:

$$\texttt{memfree}(k, n) =_{\text{dfn}} (\mu(k) = \bot) \wedge \ldots \wedge (\mu(k + n - 1) = \bot)$$

During computation, three kind of errors can happen. Namely, (i) accessing an unallocated memory cell (MemError), (ii) lacking free memory to perform a `new` instruction (MemFull) or (iii) freeing an already unallocated memory cell (FreeError).

The (MemError) rule means that there is a mechanism (either internal or external via system calls performed when memory is read) able to detect dangling pointers. This, of course, is a rather costly operation at runtime and quite hard to perform on static code. On the other hand, memory leaks are not detected.

Both these issues can be discarded if we consider that our programs are obtained by compilation of a higher level language with efficient memory management and garbage collection. In the following, we will assume that programs are *safe* in the sense that they never cause one of the error rules to be triggered.

Definition 2. A *configuration* is a triplet $\theta = \langle \texttt{IP}, \mu, \sigma \rangle$ where the *instruction pointer* `IP` is a label, μ is a heap and σ is an environment. A configuration is *final* if it's an error or $\texttt{IP} = \texttt{end}$. Θ is the set of configurations and Θ_{err} is the set containing configurations and the errors that may happen during computation.

Definition 3. Programs are executed using a straightforward small-step semantics whose rules are described in Figure 2. The relation $p \vdash \theta_1 \xrightarrow{t} \theta_2$ means that the new configuration is θ_2 after executing the instruction $t = \texttt{instr}(\texttt{IP})$ where `IP` is the instruction pointer of θ_1.

An *execution* of a program p is a sequence of configurations computed step by step. We write $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$ if $p \vdash \theta_i \xrightarrow{t_i} \theta_{i+1}$ for each $i = 0, n - 1$.

Programs compute functions from heaps to heaps as soon as certain conditions on the initial environment (with respect to the initial heap) are respected. The program of Example 1 reverses a list assuming that the heap is really the encoding of a list and that $\mathbf{r}_0$ initially contains the address of the first cell of the list.

3.3 Measuring heap Usage

In order to control space resources, we consider the number of new heap cells allocated during any execution of a program. We do not consider the number of cells of the input. So, only the "workspace" is counted like in read-only model of computation.

If we want to control the real space usage, we have to take values into account. Indeed, working with unbounded values means that one needs logarithmic space to actually store them. To constantly

Instruction	$\mathbf{r}_0$	$\mathbf{r}_1$	**next**	**val**	**tmp**	Heap										Heap size $\lvert\theta\rvert$
	2					$\perp$	1	9	$\perp$	3	0	$\perp$	$\perp$	2	5	6
next $:= 0$; `jz`	2		**0**			$\perp$	1	9	$\perp$	3	0	$\perp$	$\perp$	2	5	6
val $:= *\mathbf{r}_0$	2		0	**1**		$\perp$	1	9	$\perp$	3	0	$\perp$	$\perp$	2	5	6
tmp $:= \mathbf{r}_0 + 1$	2		0	1	**3**	$\perp$	1	9	$\perp$	3	0	$\perp$	$\perp$	2	5	6
`free` $\mathbf{r}_0$			0	1	3	$\perp$	$\perp$	9	$\perp$	3	0	$\perp$	$\perp$	2	5	**5**
$\mathbf{r}_0 :=$ ***tmp**	**9**		0	1	3	$\perp$	$\perp$	9	$\perp$	3	0	$\perp$	$\perp$	2	5	5
`free` **tmp**	9		0	1		$\perp$	$\perp$	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	**4**
$\mathbf{r}_1 :=$ `new` 2	9	**1**	0	1		**0**	**0**	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	**6**
$*\mathbf{r}_1 :=$ **val**	9	1	0	1		**1**	0	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	6
$\mathbf{r}_1 ++$	9	**2**	0	1		1	0	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	6
$*\mathbf{r}_1 :=$ **next**	9	2	0	1		1	**0**	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	6
$*\mathbf{r}_1 --$	9	**1**	0	1		1	0	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	6
next $:= \mathbf{r}_1$; `jz`	9	1	**1**	1		1	0	$\perp$	$\perp$	3	0	$\perp$	$\perp$	2	5	6
val, $\mathbf{r}_0 :=$ `destr`$(\mathbf{r}_0)$	**5**	1	1	**2**		1	0	$\perp$	$\perp$	3	0	$\perp$	$\perp$	$\perp$	$\perp$	**4**
$\mathbf{r}_1 :=$ `cons`(**val**, **next**)	5	**4**	1	2		1	0	**2**	**1**	3	0	$\perp$	$\perp$	$\perp$	$\perp$	**6**
$\mathbf{r}_1 --$	5	**3**	1	2		1	0	2	1	3	0	$\perp$	$\perp$	$\perp$	$\perp$	6
next $:= \mathbf{r}_1$; `jz`	5	3	**3**	2		1	0	2	1	3	0	$\perp$	$\perp$	$\perp$	$\perp$	6
val, $\mathbf{r}_0 :=$ `destr`$(\mathbf{r}_0)$	**0**	3	3	**3**		1	0	2	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	**4**
$\mathbf{r}_1 :=$ `cons`(**val**, **next**)	0	**6**	3	3		1	0	2	1	**3**	**3**	$\perp$	$\perp$	$\perp$	$\perp$	**6**
$\mathbf{r}_1 --$	0	**5**	3	3		1	0	2	1	3	3	$\perp$	$\perp$	$\perp$	$\perp$	6
next $:= \mathbf{r}_1$; `jz` ; `end`	0	5	**5**	3		1	0	2	1	3	3	$\perp$	$\perp$	$\perp$	$\perp$	6

Figure 1: An example run of the reverse program.

keep this into mind, we speak of *heap complexity* rather than the usual *space complexity*.

Of course, in any real computer, the set of values is bounded (*e.g.* , it is the set `long int` of 32 bits integers). In this case, the real space usage is directly proportional to the heap usage, heap and space complexity are the same up to a multiplicative constant.

Definition 4. Let $\theta = \langle \texttt{IP}, \mu, \sigma \rangle$ be a configuration. The *heap size* $|\theta|$ of θ is card$\{n, \mu(n) \neq \perp\}$, that is the number of non-$\perp$ elements in μ.

The heap measure is a function such that for any program p, and any configuration θ_0

$$\mathit{heap}_p(\theta_0) = \begin{cases} \max_{i \in 0 \ldots n}\{|\theta_i| - |\theta_0|\} & \text{if } p \vdash \theta_0 \xrightarrow{t_1} \theta_1 \ldots \xrightarrow{t_n} \theta_n \\ & \text{and } \theta_n \text{ final} \\ \perp & \text{otherwise} \end{cases}$$

In other words, the *heap usage* $\mathit{heap}_p(\theta_0)$ is the maximal heap size of an encountered configuration when the computation terminates.

Given a total function $f : \mathbb{N} \to \mathbb{N}$, we define the set of heap-bounded programs:

$$\mathit{Heap}(f) = \{p \mid \mathit{heap}_p(\theta) \leq f(|\theta|) \forall \theta, \mathit{heap}_p(|\theta|) \neq \perp\}$$

Definition 5. LINHEAP is the class of programs running with a linear heap. That is LINHEAP $= \bigcup Heap(f)$ where the union ranges over all $f : x \mapsto \beta \times x + \alpha$.

NSI is the class of programs running with at most a constant increase in heap size, that is NSI $= \bigcup Heap(f)$ where the union ranges over all $f : x \mapsto \alpha$.

4. RESOURCE PETRI NET

4.1 Petri Nets in a nutshell

We briefly describes Petri nets and reader may consult the survey of Murata [25] for more details. A Petri net N is a quadruplet (S, T, A, ω) where S is a set of *places*, T is a set of *transitions* and A is a set of arrows such that (S, T, A) is a bipartite oriented graph. That is, S and T are disjoint and each arrow is either from S to T or from T to S. An arrow between s and t (t and s) has a weight $\omega(s,t)$ ($\omega(t,s)$). As usual, places are graphically represented by circles while transitions are represented by squares.

A *marking* M assigns to each place s a natural number $M(s)$ which indicates the number of *tokens* in it.The pre-set of a transition t is ${}^\bullet t = \{s \in S / (s,t) \in A\}$. The post-set of a transition t is $t^\bullet = \{s \in S / (t,s) \in A\}$.

A transition t is *enabled* at marking M if and only if for any place s in ${}^\bullet t$, $M(s) - \omega(s,t) \geq 0$. It can then be *fired* thus reaching marking M':

$$M'(s) = M(s) - \omega(s,t) + \omega(t,s)$$

We write $N \vdash M \xrightarrow{t} M'$, $N \vdash M_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_n$ is a *run* of the Petri net.

4.2 Control Flow Petri Net

Definition 6. Let p be a program, its *control flow Petri net* (CFPN) *cfpn*(p) is defined as follows:

- The set of places is exactly the set $\mathcal{L}$ of labels of p.
- For each instruction that is neither `jz` nor `end`, there is a transition[1] bearing the same name and for each `jz` **r** `lbl` `lbl'` instruction, there are two transitions $\mathbf{r} = 0$ and $\mathbf{r} \neq 0$.
- For each label `lbl` such that `instr`(`lbl`) $\notin \{$`jz`, `jmp`, `end`$\}$, there is one arrow from `lbl` to `instr`(`lbl`) and one from `instr`(`lbl`) to `next`(`lbl`).

[1]Formally, a transition is a pair composed of a label and the instruction name.

$$\frac{\texttt{instr(IP)} = \mathbf{r}_1, \cdots, \mathbf{r}_l := \texttt{op}(\mathbf{r}'_1, \cdots, \mathbf{r}'_k) \quad v'_j = \sigma(\mathbf{r}'_j)_{j \in 1 \ldots k} \quad v_1, \cdots, v_l = [\![\texttt{op}]\!](v'_1, \cdots, v'_k)}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{next(IP)}, \mu, \{\mathbf{r}_i \leftarrow v_i\}_{i \in 1 \ldots l} \sigma \rangle} \text{ Operator}$$

$$\frac{\texttt{instr(IP)} = \mathbf{r} := *\mathbf{r}' \text{ or } \texttt{instr(IP)} = *\mathbf{r}' := \mathbf{r} \quad \mu(\mathbf{r}') = \bot}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \text{MemError}} \text{ MemError}$$

$$\frac{\texttt{instr(IP)} = \mathbf{r} := *\mathbf{r}' \quad \mu(\mathbf{r}') = v}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{next(IP)}, \mu, \{\mathbf{r} \leftarrow v\} \sigma \rangle} \text{ Load}$$

$$\frac{\texttt{instr(IP)} = *\mathbf{r} := \mathbf{r}' \quad \mu(\mathbf{r}) \in \mathcal{V}}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{next(IP)}, \{\sigma(\mathbf{r}) \leftarrow \sigma(\mathbf{r}')\} \mu, \sigma \rangle} \text{ Store}$$

$$\frac{\texttt{instr(IP)} = \mathbf{r} := \texttt{new}\, n \quad \forall k \in \mathbb{N}, \neg \texttt{memfree}(k, n)}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \text{MemFull}} \text{ MemFull}$$

$$\frac{\texttt{instr(IP)} = \mathbf{r} := \texttt{new}\, n \quad \exists k \in \mathbb{N}, \texttt{memfree}(k, n)}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{next(IP)}, \{k \leftarrow 0, \ldots, k + n - 1 \leftarrow 0\} \mu, \{\mathbf{r} \leftarrow k\} \sigma \rangle} \text{ Alloc}$$

$$\frac{\texttt{instr(IP)} = \texttt{free}\, \mathbf{r} \quad \mu(\mathbf{r}) = \bot}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \text{FreeError}} \text{ FreeError}$$

$$\frac{\texttt{instr(IP)} = \texttt{free}\, \mathbf{r} \quad \mu(\mathbf{r}) \neq \bot}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{next(IP)}, \{\sigma(\mathbf{r}) \leftarrow \bot\} \mu, \{\mathbf{r} \leftarrow 0\} \sigma \rangle} \text{ Free}$$

$$\frac{\texttt{instr(IP)} = \texttt{jmp lbl}}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\texttt{instr(IP)}} \langle \texttt{lbl}, \mu, \sigma \rangle} \text{ Jump}$$

$$\frac{\texttt{instr(IP)} = \texttt{jz}\, \mathbf{r}\, \texttt{lbl}\, \texttt{lbl}' \quad \sigma(\mathbf{r}) = 0}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\mathbf{r}=0} \langle \texttt{lbl}, \mu, \sigma \rangle} \text{ IfTrue}$$

$$\frac{\texttt{instr(IP)} = \texttt{jz}\, \mathbf{r}\, \texttt{lbl}\, \texttt{lbl}' \quad \sigma(\mathbf{r}) \neq 0}{p \vdash \langle \texttt{IP}, \mu, \sigma \rangle \xrightarrow{\mathbf{r} \neq 0} \langle \texttt{lbl}', \mu, \sigma \rangle} \text{ IfFalse}$$

Figure 2: Small-steps semantics

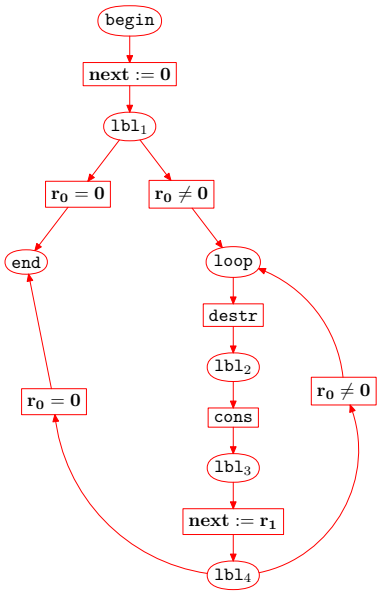

Figure 3: CFPN of the reverse program (with operators)

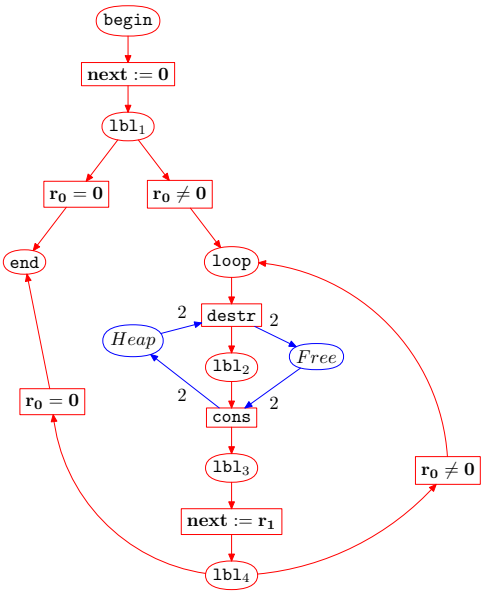

Figure 4: RPN of the reverse program (with operators)

- For each label `lbl` such that `instr(lbl) = jmp lbl'`, there is one arrow from `lbl` to `instr(lbl)` and one from `instr(lbl)` to `lbl'`.
- For each label `lbl` such that `instr(lbl) =jz r lbl`$_0$ `lbl`$_1$, there are the four arrows $(\texttt{lbl}, \mathbf{r} = 0)$, $(\mathbf{r} = 0, \texttt{lbl}_0)$, $(\texttt{lbl}, \mathbf{r} \neq 0)$ and $(\mathbf{r} \neq 0, \texttt{lbl}_1)$.

Both places and transitions are named after the label or instruction they represent. No distinction are made between a place or transition and the label or instruction as long as the context is clear. The CFPN of the reverse program is displayed on Figure 3. If we only consider the places, we obtain the usual Control Flow Graph (CFG) of the program. The CFG of the reverse program is on Figure 5.

If $\theta = \langle \texttt{IP}, \mu, \sigma \rangle$ is a configuration of a program p, the corresponding marking M^θ of *cfpn*(p) is the marking that puts one token in the place `IP` and none in the other places.

PROPOSITION 2. *Let p be a program. If $p \vdash \theta_0 \xrightarrow{t_1} \dots \xrightarrow{t_n} \theta_n$ is an execution of p, then cfpn$(p) \vdash M^{\theta_0} \xrightarrow{t_1} \dots \xrightarrow{t_n} M^{\theta_n}$ is a run of cfpn(p).*

4.3 Resource Petri Net

We now turn to represent the heap memory mechanism by a Resource Petri Net (RPN). In order to define a RPN, we add to CFPN construction described above two new places *Heap* and *Free*. As previously, each program execution is mapped on a RPN run which, unlike the previous CFPN model, is resource-aware. For this, a token represents a heap cell. At the beginning of a program execution, the place *Free* contains the number of free cells. When a program requests n new cells, then we need to have n tokens in the place *Free*. In this case, we put n tokens in the place *Heap*. Otherwise, there is not enough free memory, and the run of the RPN stalls. The number of tokens in the place *Heap* is the number of cells which have been allocated during a program execution.

Inversely, when a `free` **r** instruction is executed, then one token is moved from the *Heap* place to the *Free* one. Thus we recycle discarded heap cells.

So, a RPN is a simple model which allows to simulate executions of programs and to check memory usage at the same time. The *Heap* place should contain as many tokens as the current size of the heap (the number of non-$\perp$ cells). The *Free* place should contain as many tokens as there is free memory (this depends on the machine used, not only on the program).

Definition 7. Let p be a program. The *resource Petri net rpn*(p) of p is obtained by adding to *cfpn*(p) two places *Free* and *Heap* and four kind of arrows

- An arrow from the *Free* place to each `new` n transition of weight n.
- An arrow from each `new` n transition to *Heap* place of weight n.
- An arrow from each `free` transition to the *Free* place of weight 1
- An arrow from the *Heap* place to each `free` transition of weight 1.

The above construction depends on the resource amount requested by each instruction, which may be measured by a weight:

$$\omega(\texttt{new}\ n) = -n \qquad \omega(\texttt{free}\ \mathbf{r}) = 1$$

$$\omega(\texttt{instr}) = 0 \quad \text{for any other instruction}$$

The construction of RPN is based on the weight of each instruction. We generalise RPN to operators by assigning a weight, which is defined as the sum of the weights of the instructions that compose them. In this way, we have $\omega(\texttt{cons}) = -2$ and $\omega(\texttt{destr}) = 2$. We add an edge from each transition labelled by an instruction `instr` of non-zero weight to *Heap* and *Free* places. The weight

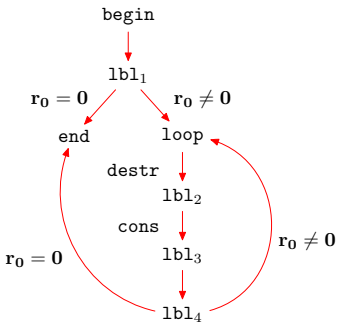

Figure 5: Control Flow Graph of the reverse program. Edges are labelled after the instruction they represent.

of this edge is the absolute value of $\omega(\texttt{instr})$ and the direction of it is given by the sign of the weight. Figure 4 illustrates this by showing the RPN of the reverse program.

Tokens in the *Heap* and *Free* places correspond to Hofmann's diamonds ($\diamond$), that is a fixed amount of space in the heap. Each token represents a diamond, either used by (one of the variable of) the program if in the *Heap* place or available if in the *Free* place. Memory allocation (such as `cons`), needs diamonds in order to be performed while memory de-allocation (such as `destr`) releases diamonds to be used later.

LEMMA 1. *In each run of a RPN, the total number of tokens in the Free and Heap places is constant.*

To each configuration $\theta = \langle \texttt{IP}, \mu, \sigma \rangle$ of p, we associate a marking M_i^θ of *rpn*(p):

$$M_i^\theta(\textit{Heap}) = |\theta| \qquad M_i^\theta(\textit{Free}) = i$$
$$M_i^\theta(\texttt{IP}) = 1 \qquad M_i^\theta(\texttt{lbl}) = 0 \qquad \text{otherwise}$$

LEMMA 2. *Let p be a program, θ be a configuration such that $p \vdash \theta \xrightarrow{t} \theta'$ and i be an integer such that $i + \omega(t) \geq 0$. Then, rpn$(p) \vdash M_i^\theta \xrightarrow{t} M_{i+\omega(t)}^{\theta'}$.*

PROOF. Consider that $t = \texttt{new}\ n$. We can remove n tokens from *Free* because $i + \omega(t) = i - n \geq 0$. So, we can fire the `new` n transitions.

Next, consider $t = \texttt{free}\ \mathbf{r}$. The place *Heap* contains $|\theta|$ tokens and by hypothesis $p \vdash \theta \to \theta'$, so we know that $|\theta| > 0$[2]. Therefore, we can remove one token from the place *Heap* and fire the transition. □

PROPOSITION 3. *Let p be a program and let N be its RPN. Assume that $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$. Let i_0 be a natural number such that $\forall k \leq n, \quad i_0 + |\theta_0| - |\theta_k| \geq 0$. Then, there is a run $N \vdash M_{i_0}^{\theta_0} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_{i_n}^{\theta_n}$ which satisfies $i_k = i_0 + |\theta_0| - |\theta_k|$ for each $k = 0, \ldots, n$.*

5. CHARACTERIZATION OF HEAP COMPLEXITY

[2]The condition $|\theta| - \omega(t) \geq 0$ is not necessarily here.

5.1 Heap complexity and resource Petri nets

THEOREM 3. *Let $f : \mathbb{N} \to \mathbb{N}$ be a total function. Let p be a program.*

$p \in$ Heap$(f(x))$ if and only if for each initial configuration θ_0 and for each execution $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$, we have rpn$(p) \vdash M_{f(|\theta_0|)}^{\theta_0} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_{i_n}^{\theta_n}$ for some $i_1, \ldots, i_n$.

The fact that the run exists means that $f(|\theta_0|)$ is a sufficiently large number of tokens to put in *Free* to avoid any deadlock.

PROOF. If for each initial configuration θ_0 and each execution $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$ there is a corresponding run

$$\textit{rpn}(p) \vdash M_{f(|\theta_0|)}^{\theta_0} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_{i_n}^{\theta_n}$$

then for any k,

$$|\theta_k| = M_{i_k}^{\theta_k}(\textit{Heap}) \leq M_{f(|\theta_0|)}^{\theta_0}(\textit{Heap}) + M_{f(|\theta_0|)}^{\theta_0}(\textit{Free})$$
$$= |\theta_0| + f(|\theta_0|)$$

Conversely, deadlocks in the run may come from three sources: the label places, the *Heap* place or the *Free* place. The label places cannot cause deadlock due to Proposition 2. The *Heap* place may only cause deadlock if a `free` transition should be fired but the *Heap* place is empty. However, this would correspond to executing a `free` instruction on an empty heap, thus triggering the (FreeError) rule.

The *Free* place can cause deadlock if a `new` n transition has to be fired but *Free* contains less than n token. However, Prop. 3 claims that if we start from marking $M_{i_0}^{\theta_0}$ such that for all $1 \leq k \leq n$, $i_0 + |\theta_0| - |\theta_k| \geq 0$ there will be no deadlock. Since $p \in$ *Heap*(f), $|\theta_k| \leq f(|\theta_0|) + |\theta_0|$ and thus $i_0 + |\theta_0| - |\theta_k| \geq i_0 - f(|\theta_0|)$. So by starting from $M_{f(|\theta_0|)}^{\theta_0}$, there will be no deadlock. □

Notice that the (implicit) quantifier on the Petri net runs is existential and not universal. To each execution of the program corresponds a run, but some runs correspond to no execution of the program (because the Petri net does not mimic the semantics tightly, especially tests). This means that unless one can decide which run corresponds to an execution, analysis of the Petri net will not capture every programs (because in some cases, the "invalid" runs do not correspond to an execution). This is a necessarily lose if we want our criterion to be decidable, as mentioned in the discussion at the end of Section 1.2.

COROLLARY 2. *$p \in$ NSI if and only if there exists a constant α such that for each initial configuration θ_0 and each execution $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$, we have cfpn$(p) \vdash M_\alpha^{\theta_0} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_{i_n}^{\theta_n}$ where $i_k = |\theta_0| + \alpha - |\theta_k|$.*

$p \in$ LINHEAP if and only if there exists α and β such that for each initial configuration θ_0 and each execution $p \vdash \theta_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} \theta_n$, we have cfpn$(p) \vdash M_{\beta \cdot |\theta_0| + \alpha}^{\theta_0} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_{i_n}^{\theta_n}$ where $i_k = (\beta + 1)|\theta_0| + \alpha - |\theta_k|$.

5.2 Detection of non-size increasing programs

THEOREM 4. *There is a polynomial time procedure to detect if a program is NSI.*

PROOF. Let p be a program. We construct a directed graph G from *rpn*(p) that we name the *resource control graph* of p as follows. Nodes are label place. There is an edge from `lbl` to `lbl`$'$

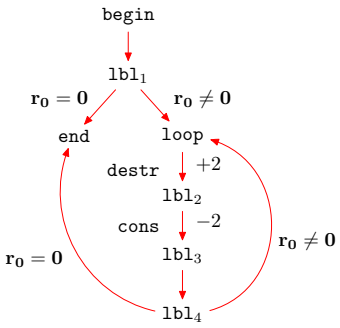

Figure 6: Resource Control Graph of the reverse program. Edges are labelled after the instruction they represent.

if there is a transition from `lbl` to `lbl`$'$ in $rpn(p)$. The weight assigned to an edge is the weight of the corresponding transition, that is the weight $\omega(\texttt{instr}(\texttt{lbl}))$. In other words, the resource control graph is the control flow graph where each arrow as the same weight as the corresponding instruction.

If the resource control graph contains no cycle of strictly negative weight, then the program is in NSI. This criterion can be decided in polynomial time via Bellman-Ford's algorithm.

Indeed, any path in the resource control graph correspond to a run in the RPN. The weight of the path being exactly the number of token removed from (if negative) the *Free* place at the end of the run. If there is no cycle of negative weight, then the minimum weight of all paths, α, is well defined. So, no run can remove more than α token from *Free*. Hence, each execution corresponds to a run that can be performed with $\max(-\alpha, 0)$ token initially in *Free* and p is NSI.

□

The above demonstration provides also the number of cells which are necessary to perform any computation. Therefore, we may be able to set up a proof procedure, like in proof carrying code paradigm, in polynomial time to calculate and certify that a program uses a fixed amount of heap.

Hofmann has shown that Non Size Increasing programs can be compiled into `C` without using `malloc` [13]. That is, in our case, we can write an equivalent program without the `new` instruction, or, more precisely, starting with a single `new` α instruction during the initialization process. Amadio already has a PTIME bound to detect NSI programs in terms of quasi-interpretation [3]. And linear space usage can also be characterized by quasi-interpretations [8].

Lasltly, notice that a cycle of positive weight may cause damages by releasing potentially non-allocated heap cells. We do not control this kind of troubles here.

Example 2. The resource graph of the reverse program is given in Figure 6. Since there is no cycle of negative weight, so the program is NSI. Moreover, the weight of any path starting from `begin` is either 0 or $+2$, so choosing $\alpha = 0$ is enough, meaning that the program doesn't use more memory that initially allocated.

Of course, the above method does not capture all non size increasing programs. For example, if in a loop like (written in a `C`-like syntax for readability):

$$\text{for}(i = 1, i \leq \mathbf{r}, i++)\{\mathbf{r}' := \texttt{new}\ 1;\}$$

the method fails. However, we can roughly distinguish three cases for this kind of loops.

- $\mathbf{r}$ is an immediate value, that is the loop actually is, say,

  $$\text{for}(i = 1, i \leq 10, i++)\{\mathbf{r}' := \texttt{new}\ 1;\}$$

  In this case, the allocation can be performed before the loop:

  $$\mathbf{r} := \texttt{new}\ 10; \text{for}(i = 1, i \leq 10, i++)\{\mathbf{r}' := \mathbf{r} + i;\}$$

  This allows to have the same effect but put the allocation out of the loop.

- $\mathbf{r}$ is a value stored in a register which happen to be a constant. In this case, constant propagation allows to detect at compile time that $\mathbf{r}$ can be replaced by an immediate value.

- $\mathbf{r}$ is the value of a register which depends on the inputs. In this case, the program is probably not NSI because it perform a number of allocations dependant on the inputs.

Both the first two transformations can be performed automatically at compile time, just before the analysis is performed. Moreover, construction of the CFG and RCG can help find potentially harmful loops (those with a positive weight) and thus help the transformation of these loops. Of course, programs where constant propagation fails or programs with loops of the third kind will not be detected as NSI. Hopefully, there are few of them among the commonly used algorithms...

It is also important to notice that any algorithm that fits into Hofmann's framework can be detected as NSI by our method. Indeed, Hofmann provides a way to compile NSI programs into `malloc`-free programs and our method obviously detect `malloc`-free programs as NSI (since they do not allocate memory, the RCG has no edge of negative weight and cannot have cycle of negative weight).

6. OTHER CHARACTERIZATIONS

6.1 Linear Heap Space

The detection of linear space computation is closely related to the one of non size increasing computation, and the previous method can be modified to handle linear heap run. The idea is that each input cell gives β new cells when it is deallocated, instead of one in the case of a non-size increasing memory allocation. On the other hand, cells which are allocated during an execution only count for one. To perform this analysis, we consider two kinds of heaps: an *input heap* that cannot be further re-allocated ; and a *work heap*, initially empty, that can be allocated. Then, programs have two `free` instructions: $\texttt{free}_i$ to free input cells and $\texttt{free}_w$ to free work cells[3]. The `new` only allocates memory on the work heap μ_w.

Now, we extend RPN in order to take into account both new free instructions by assigning the following weights to them:

$$\omega(\texttt{free}_i\ \mathbf{r}) = \beta \qquad \omega(\texttt{free}_w\ \mathbf{r}) = 1$$

THEOREM 5. *There is a polynomial time procedure to detect if a program is linear heap computable.*

PROOF. The proof is similar to the demonstration of Theorem 4. We need to find β such that the resource control graph has no cycles of strictly negative weight. Notice that this will also give us the values β and α and a precise bound on the space usage. □

[3]We cope with the difficulty to detect which free instruction releases memory from the input or work heap. A way is to restrict the use of input register like in [15].

Again, this decision procedure does not capture all linear heap programs but again use of constant propagation may help.

6.2 Logarithmic Heap Space

Usual definitions of logarithmic space computations often present it by saying that we are allowing to have a number of pointers into the initial data but not to alter it. In our case, registers which only store addresses in the heap are clearly pointers while some others (such as the **val** in the reverse program) are not. So if we have a NSI-program and we can control the size (that is the value) of registers, then it will be possible to compute it in logarithmic space.

In order to do this, we must be careful about heap management. Indeed, pointers (that is registers containing addresses in the heap) might have a value as big as the address of any allocated heap-cell. But this value might well be unrelated to the size of the heap (that is the number of allocated cells) just because the allocated cells might be non-contiguous.

So we need to somehow force the heap to be as contiguous as possible. The initial heap must respect these conditions and the `memfree` predicate also. In real computer, these conditions may be ensured by having a set of virtual memory addresses to access the heap and those virtual addresses are translated into real memory addresses by the system via a map between the virtual memory of each process and the real memory. The `memfree` system call has access to these maps and in addition to allocating some real memory it extends the map of the process.

Keeping in mind the fact that heap is contiguous, we now can compute NSI-programs in logarithmic space as soon as the values stored in registers is bound by the (initial) size of the heap at each step of the computation.

6.3 Lack of Resources

Since it is not possible to free more memory that what is allocated, no execution can free an unbounded number of cells. So, any potentially infinite execution whose corresponding run in the RPN (resp. path in the RCG) removes infinitely many tokens from *Heap* (resp. has weight $+\infty$) will stop once the heap is empty and does not correspond to a *real* infinite execution.

Similarly, on any real computer, the available memory is finite and bounded. So, any potentially infinite execution whose corresponding run in the RPN (resp. path in the RCG) removes infinitely many tokens from *Free* (resp. has weight $-\infty$) will stop once all the memory has been allocated and does not correspond to any *real* infinite execution. This leads to a simple criterion to detect termination by lack of resources.

PROPOSITION 4. *Let p be a program and G be its RCG. If all the infinite paths of G have infinite weight ($\pm\infty$), then all executions of the program terminate by lack of resources.*

This approach is somewhat similar to the Size Change Principle (SCP) [17] in the sense that we check if a given finite resource (in our case memory) is exhausted during computation. However, SCP is much more efficient that our simple criterion since it takes into account the values of the variables and not only the global amount of memory. Typically, this lack of resources criterion is unable to detect termination of the reverse program while SCP detects it by checking the length of the list.

7. CONCLUSION

The main result that we have presented is a (incomplete) criterion to determine if a low-level program runs within a fixed amount of heap memory, and to calculate an upper bound on this heap size. Moreover, this criterion is polynomial time computable. One of the main advantages of our approach compared with the other ones that we have cited in the text, is that we are dealing with a low-level programming language without type annotations. Aspinall and Compagnoni [5] derived a similar characterisation for assembly language but still by using the type discipline originally introduced by Hofmann [14]. We believe that the representation in term of Resources Petri Nets shed some light on Hofmann's diamond-concept. In particular, we do not need any more the linearity conditions introduced in these works. Moreover, we do not forbid dynamic heap allocation.

More generally, our method shows that it is quite easy to control the size of any buffer. In the same way that we control the size of the heap via the `new` and `free` instructions, we could similarly control depth of a data stack (via `push` and `pop`), of a control stack (via `call` and `return`), or the value of a counter (via increment and decrement). We may also deal with several kinds of tokens, which represent different amount of memory. A challenging direction is to compare our (more) theoretical method with the exciting approaches of [27, 9]. Another interesting direction would be to incorporate our methods into the Hoare or Dynamic logic paradigm. For this, we could follow the recent works of Leivant [18, 19].

Finally, we think that our method can be extended in order to control the size of each variable of a program independently. We aim at having a criterion for termination similar to the Size Change Principle. We have started to do so on simple basis in [24, 21] and a bit more elaborated in [22]. We intend to go further in this direction.

8. REFERENCES

[1] A. Abel and T. Altenkirch. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[2] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of Non-Size Increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00)*, pages 84–91, 2000.

[3] R. Amadio. Max-plus quasi-interpretations. In *TLCA*, 2003.

[4] R. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, 2004. To appear.

[5] D. Aspinall and A. Compagnoni. Heap Bounded Assembly Language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code)*, 31:261–302, 2003.

[6] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[7] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation: a way to control ressources. *Theoretical Computer Science*. under revision.

[8] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-Interpretations and Small Space Bounds. In *Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164, April 2005.

[9] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt-driven programs. In *SAS 03: Static Analysis*, Lecture Notes in Computer Science 2694, pages 109–126. Springer-Verlag, 2003.

[10] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the*

International Conference on Logic, Methodology, and Philosophy of Science, pages 24–30. North-Holland, Amsterdam, 1962.

[11] L. Colson. Functions versus Algorithms. *EATCS Bulletin*, 65, 1998. The logic in computer science column.

[12] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[13] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[14] M. Hofmann. The strength of Non-Size Increasing computation. In *Proceedings of POPL'02*, pages 260–269, 2002.

[15] N. Jones. The expressive power of higher order types or, life without cons. *Journal of Functional Programming*, 11(1):55–94, 2000.

[16] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967. Reprint, MIT press 1987.

[17] C. S. Lee, N. D. Jones, and A. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL'01*, volume 28, pages 81–92. ACM press, January 2001.

[18] D. Leivant. Proving Termination Assertions in Dynamic Logics. In *Proceedings of LICS'04*, pages 89–98, 2004.

[19] D. Leivant. Matching explicit and modal reasoning about programs: a proof theoretic delineation of dynamic logic. Technical report, 2005.

[20] D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, 19(1,2):167–184, September 1993.

[21] J.-Y. Marion and J.-Y. Moyen. Termination and ressource analysis of assembly programs by Petri Nets. Technical report, Loria, 2003.

[22] J.-Y. Marion and J.-Y. Moyen. Termination and non size increasingness of assembly programs. In *AppSem*, 2005.

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

[24] J.-Y. Moyen. *Analyse de la complexité et transformation de programmes*. Thèse d'université, Nancy 2, Dec 2003.

[25] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[26] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*. To appear.

[27] J. Regehr. Say no to stack overflow. *Embedded systems programming*, 2004.

[28] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.