Termination and resource analysis of assembly programs by Petri nets

Jean-Yves Marion & Jean-Yves Moyen

LORIA - Calligramme project
615, rue du jardin botanique
54602 Villers-lès-Nancy, France
and
INPL-Ecole des Mines de Nancy
Parc de Saurupt,
54042 Nancy CEDEX, France
{Jean-Yves.Marion, Jean-Yves.Moyen}@loria.fr

Abstract

We investigate resource analysis of assembly programs. For this, we construct a Petri net interpretation of programs and define a resource termination principle which is in co-NP. Then, we give a criterion to determine if a program runs without new memory (non size increasing property) which is in Ptime.

1 Introduction

The "ressource termination" principle is an attempt to prove automatically certain computational ressource properties on assembly programs. We focus (i) on termination and (ii) on on non-size increasing computations.

Temination. The ressource termination principle is based on a program interpretation by mean of Petri nets. We call them change size Petri net. A program terminates on all inputs if every firing sequence leads to a deadlock. The intuition is that a program terminates when there is no more ressource to consume. At the present time, we just consider that the programm ressource are the data stored in variables. The Petri net interpretation keeps track of every ressource variation in a global way which stops when ressources are exhausted. So it gives a kind of mechanic interpretation of the execution of a program.

The work on termination is inspired by the Size Change Termination Principle of Lee, Jones and Ben-Amram [5]. There are three main differences. First, we study imperative language and not first order functional language. Second, the class of programs that we succesfully consider is not the same, like Euclide gcd algorithm or the quicksort. Third, we establish that the ressource termination criterion is in co-NP.

Non-size increasing computations. A program is non-size increasing if the program runs without new memory allocation on all inputs. The concept of non-size increasing computation was introduced by Hofmann [3]. We extend size change Petri nets to nets with a memory device that we call Ressource Petri nets. A program is non-size increasing if each firing sequence from an initial marking leads to a final marking. We establish that this property is decidable in polynomial time.

Of course, it is important to establish that a program terminates. In fact, we are not aware of attempts to prove automatically termination on imperative languages. The reader may consult [5] for a short survey on termination methods applied to programming. But, the main reason of this work is to control computational ressources. Indeed, an important safety property in embedded systems or mobile codes is that the memory usage is controled. There are several reasons including the detection of memory overflow, to avoid denial of service attack, or to compile more efficiently programs [4]. It also worth is noticing that a termination proof by "size change principle" can use the fact that some recursive call is non-size increasing.

2 Programs

2.1 Syntax and useful definitions

We consider a assembly-like programming language L. The set of data-value is the set of natural numbers. The syntax of the programs is defined by the following rules:

$$\begin{array}{rcl}
\texttt{pgm} & ::= & \texttt{input}: i_1, \cdots, i_m; \texttt{output}: o_1, \cdots, o_n; \\
& & 0 : \texttt{inst}_0; \ldots; n : \texttt{inst}_n; end \\
\texttt{inst} & ::= & \texttt{dec}\, x \mid \texttt{inc}\, x \mid \texttt{jmp}\, \texttt{lbl}_i \mid \texttt{jz}\, x\, \texttt{lbl}_i \mid \\
& & \texttt{call}\, \texttt{lbl}_i \mid \texttt{return} \\
x & ::= & \texttt{variable identifier} \\
\texttt{lbl} & ::= & 0 | \ldots | n | end
\end{array}$$

The input variables are $i_1, \cdots, i_m$ and the output variables are $o_1, \cdots, o_n$. The sets of input and output variables are not necessarily disjoint. x represents any variable, either input, output or new variable just used during the computation. A program computes a function which has a fixed arity and co-arity, that is from $\mathbf{N}^m$ to $\mathbf{N}^n$. The `inc` (resp. `dec`) increments (resp. decrements) a variable. The `jmp` instruction jumps to the specified label. The `jz` instruction jumps to the label only if the variable is equal to zero, otherwise the next instruction is executed. The `call` instruction jumps to the label and resume the computation when a `return` instruction is met. The `call`/`return` mechanism allows to divide a program into subroutines. In order to avoid confusion between `call` and `return`, a jump between subroutines is not allowed.

We define a function `instr` from labels to instructions which gives the instruction indicated by the label. That is $\mathtt{instr}(i) = \mathtt{inst}_i$.

2.2 Semantics

Programs are interpreted following the standard call-by-value semantics defined in Figure 1.

Definition 1 (Substitution). *A substitution σ is a mapping from variables to integers. The substitution $\sigma\{x \leftarrow v\}$ denotes the substitution where x is assigned to v and the other variables keep the same value with respect to σ. Similarly, $\{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_m\}$ designates the substitution which assigns the integer v_i to the variable x_i and assigns 0 to other variables.*

Definition 2 (Return stack). *A return stack $\mathtt{stk}$ is a list of labels. The empty return stack is noted $[]$.*

Definition 3 (State). *A program state is a triple $(\mathtt{lbl}, \sigma, \mathtt{stk})$ where $\mathtt{lbl}$ is a label, σ is a substitution and $\mathtt{stk}$ is a return stack, or a program state is the special state $\perp$ used to handle errors.*

When it is convenient, we use letters s, s' to denote states. Also, we just write state instead of program state if there is no possible confusion.

Definition 4 (Program semantics). *The relation $\to$ is the one step transition rule. $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$. The program pgm computes the partial function $[\![\mathit{pgm}]\!]$ defined by $[\![\mathit{pgm}]\!](v_1, \cdots, v_n) = (\sigma(o_1), \cdots, \sigma(o_n))$ if $\mathit{pgm} \vdash 0, \{i_1 \leftarrow v_1, \ldots, i_n \leftarrow v_m\}, [] \xrightarrow{*} \mathit{end}, \sigma, []$.*

The programming language L simulates step by step the counter machine introduced by Shepherdson and Sturgis [7]. As an immediate consequence, any function which is computable by a Turing machine is computable by L. For this, we have to encode strings by numbers which is done by well-known methods. More instructions could be added in L like $x_i = x_j, x_i = v$ where v is a integer, or arithmetic operations.

However, there is no access to memory say by a mechanism of indirect fetch and store. In the same way, a call does not save current variables.

The following examples will serve to illustrate our methods.

1. **Addition**.
   The following programs compute the addition of two numbers. The first program, on the left, uses no `call` and is similar to the tail-recursive version in functional programming (*i.e.* $add(x+1, y) = add(x, y+1)$). The second one, on the right, uses `call` and is similar to the non-tail recursive functional program (*i.e.* $add(x+1, y) = add(x, y) + 1$). Note that the second version uses a "wrapper" to do the first call, in order to avoid a problem when the `return` instruction is executed with an empty return stack. The *end* label has been moved in order to show clearly where is the wrapper and where is the recursive subroutine.

   Input: x, y.

   Output: y.

0 : jz x *end* 1 : dec x 2 : inc y 3 : jmp 0 *end*	0 : call 1 *end* 1 : jz x 5 2 : dec x 3 : call 1 4 : inc y 5 : return

2. **Copies**.
   We present two macros *copy* and *dup* that are very useful. The *copy* macro loses the content of the original variable. The *dup* macro performs a duplication.

$x := copy(y)$	$x := dup(y)$
0 : *jz y end* 1 : *dec y* 2 : *inc x* 3 : *jmp* 0 *end*	0 : *jz y* 5 1 : *dec y* 2 : *inc x* 3 : *inc* y' 4 : *jmp* 0 5 : *jz* y' *end* 6 : *dec* y' 7 : *inc y* 8 : *jmp* 5 *end*

3. **Greatest Common Divisor**.
   The following program computes the gcd of two numbers using Euclide's algorithm. The principle of the algorithm is that $gcd(a, b) = gcd(a - b, b)$ if $a > b$. The operation $a > b$ is assumed to be a primitive of the language. Note that the *copy* macros should be replaced by the corresponding code. Other termination analysis, such as the Size Change Principle, are unable to detect termination of Euclide's algorithm.

   Input: a, b.

   Output: a.

```
 0 : jz b end
 1 : if a < b jmp 9
 2 : jz b 7
 3 : dec b
 4 : dec a
 5 : inc tmp
 6 : jmp 2
 7 : b := copy(tmp)
 8 : jmp 0
 9 : tmp := copy(a)
10 : a := copy(b)
11 : b := copy(tmp)
12 : jmp 0
end
```

Definition 5 (State Trace). *The state trace $\theta(\mathtt{lbl}, \sigma, \mathtt{stk})$ of a state $(\mathtt{lbl}, \sigma, \mathtt{stk})$ is the current instruction. It is defined as follows:*

- *If $\mathtt{instr}(\mathtt{lbl}) \neq jz\ x\ \mathtt{lbl}'$ then $\theta(\mathtt{lbl}, \sigma, \mathtt{stk}) =_{dfn} \mathtt{instr}(\mathtt{lbl})$.*
- *If $\mathtt{instr}(\mathtt{lbl}) = jz\ x\ \mathtt{lbl}'$ and $\sigma(x) = 0$ then $\theta(\mathtt{lbl}, \sigma, \mathtt{stk}) =_{dfn} x = 0$*
- *If $\mathtt{instr}(\mathtt{lbl}) = jz\ x\ \mathtt{lbl}'$ and $\sigma(x) \neq 0$ then $\theta(\mathtt{lbl}, \sigma, \mathtt{stk}) =_{dfn} x \neq 0$*

$$\frac{\mathtt{instr(lbl)} = inc\ x}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl} + 1, \sigma\{x \leftarrow \sigma(x) + 1\}, \mathtt{stk}}\ increase$$

$$\frac{\mathtt{instr(lbl)} = dec\ x \quad \sigma(x) = 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \bot}\ error$$

$$\frac{\mathtt{instr(lbl)} = dec\ x \quad \sigma(x) \neq 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl} + 1, \sigma\{x \leftarrow \sigma(x) - 1\}, \mathtt{stk}}\ decrease$$

$$\frac{\mathtt{instr(lbl)} = jmp\ \mathtt{lbl}'}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl}', \sigma, \mathtt{stk}}\ jump$$

$$\frac{\mathtt{instr(lbl)} = jz\ x\ \mathtt{lbl}' \quad \sigma(x) = 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl}', \sigma, \mathtt{stk}}\ x = 0$$

$$\frac{\mathtt{instr(lbl)} = jz\ x\ \mathtt{lbl}' \quad \sigma(x) \neq 0}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl} + 1, \sigma, \mathtt{stk}}\ x \neq 0$$

$$\frac{\mathtt{instr(lbl)} = call\ \mathtt{lbl}'}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, \mathtt{stk} \rightarrow \mathtt{lbl}', \sigma, (\mathtt{lbl} + 1, \mathtt{stk})}\ call$$

$$\frac{\mathtt{instr(lbl)} = return}{\mathtt{pgm} \vdash \mathtt{lbl}, \sigma, [\mathtt{lbl}', \mathtt{stk}] \rightarrow \mathtt{lbl}', \sigma, \mathtt{stk}}\ return$$

Figure 1: Call by value semantics

Definition 6 (Program Trace). *A program trace from a state s_0 is a finite or infinite sequence of state traces $\omega = \omega_0\omega_1\omega_2\ldots$ such that $s_0 \to s_1 \to s_2 \to \ldots$ and for all i, we have $\theta(s_i) = \omega_i$.*

Notice that if the program on some inputs is not terminating, the length of the trace is infinite.

If $\omega = \omega_0 \cdots \omega_n$ is a finite program trace, we write $s \xrightarrow{\omega} s'$ iff $s = s_0 \to \cdots s_n = s'$ and $\theta(s_i) = \omega_i$, for $0 \le i \le n$. The state s' is uniquely defined from both the state s and the program trace ω.

3 Petri Nets

This small section is self-contained and quickly explains what a Petri net is. Notations and definition are mostly taken from Jörg Desel and Javier Esparza book [1].

3.1 Nets

Definition 7 (Net). *A **net** N is a triple (S, T, F) where S is a set of* places, *T is a set of transitions, and $F \subset (S \bigcup T) \times (S \bigcup T)$ is the flow relation. We assume that S and T are disjoint and that $F \bigcap (S \times S) = F \bigcap (T \times T) = \emptyset$. So, a net defines a directed bipartite graph whose set of nodes is $S \bigcup T$ and set of edges is F.*

Graphically, places are represented by circles or ellipses and transitions by squares or rectangles.

Definition 8 (Pre-set, Post-set). *The elements of $S \bigcup T$ are called* nodes. *Given a node a of N, the set ${}^{\bullet}a = \{b : (b, a) \in F\}$ is the **pre-set** of a and the set $a^{\bullet} = \{b : (a, b) \in F\}$ is the **post-set** of a.*

3.2 Marking and deadlocks

Definition 9 (Marking). *A **marking** of a net (S, T, F) is a mapping M from places of S to integers. A marking may be represented by the vector $(M(s_1), \ldots, M(s_n))$, where $(s_1, \cdots, s_n)$ is an arbitrary fixed enumeration of the places. Markings are partially ordered with respect to each place. That is $M \trianglelefteq M'$ if and only if for each place s, $M(s) \le M'(s)$.*

Definition 10 (Enabled transition, occurrence rule). *A marking M **enables** a transition t if for each place p in ${}^{\bullet}t$, $M(p) > O$.*

*If a transition t is enabled at a marking M, then the transition t can be **fired**. The firing of t leads to the successor marking M', noted $M \xrightarrow{t} M'$, which is defined by:*

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in^{\bullet} t \text{ and } p \notin t^{\bullet} \\ M(p) + 1 & \text{if } p \in t^{\bullet} \text{ and } p \notin^{\bullet} t \\ M(p) & \text{otherwise} \end{cases}$$

A token is removed from each place in ${}^{\bullet}t$ and a token is added in each place of $t^{\bullet}$.

Definition 11 (Occurrence sequence). *Let M be a marking of a net N. If $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ is a firing sequence, then $\omega = t_1, \cdots, t_n$ is an* occurrence sequence *leading from M to M_n and we write $M \xrightarrow{\omega} M_n$.*

A sequence ω is enabled at marking M if either there exists M' such that $M \xrightarrow{\omega} M'$ or $M \xrightarrow{\omega}$ (ω is infinite).

Definition 12 (Deadlock, starvation). *A marking M is a **deadlock** if it enables no transition. If no marking M enables an infinite firing sequence, the net is **starving***

4 Termination

4.1 Size Change Petri net

We construct the *Size Change Petri Net* (SCPN) of a program `pgm` which is a Petri net that simulates the executions of a program on each input in a non-deterministic way. The formal construction of this net is not given as it is quite obfuscated for `call`s and `return`s.

We follow the construction of Esparza given in [2]. The idea is that firing a transition corresponds to executing an instruction. There is one place for each label, one for each variable and some additional places to handle return stack (mainly one for each subroutine). There is one transition for each non-`jz` instruction and two for each jz instruction.

The $call$/$return$ mechanisms works as follows. A token is given to the label that has been called, and another one is kept, waiting for the subroutine to end. When a $return$ is executed, it enables the transition after the $call$, so the program may go on.

Figure 2 shows the SCPN of the different instructions[1]. Note how an extra transition, *loop* is added if end is the return address of a call, in order to get rid of extra tokens generated by the `call`. SCPN of `call` and `return` is shown according to the following program:

$$\begin{array}{rcl} 0 & : & \texttt{call}\ 2 \\ 1 & : & \texttt{call}\ 2 \\ end & & \\ 2 & : & \texttt{jz}\ x\ 4 \\ 3 & : & \texttt{return} \\ 4 & : & \texttt{return} \end{array}$$

Definition 13 (State marking). *A program state $s = (\mathtt{lbl}, \sigma, \mathtt{stk})$ determines a marking $\mathtt{marking}(\mathtt{lbl}, \sigma, \mathtt{stk})$ defined thus:*

$$\begin{aligned} \mathtt{marking}(\mathtt{lbl}, \sigma, \mathtt{stk})(\mathtt{lbl}) &= 1 + \#_{\mathtt{lbl}}(\mathtt{stk}) \\ \mathtt{marking}(\mathtt{lbl}, \sigma, \mathtt{stk})(\mathtt{lbl}') &= \#_{\mathtt{lbl}'}(\mathtt{stk}) \qquad \text{if } \mathtt{lbl} \neq \mathtt{lbl}'. \\ \mathtt{marking}(\mathtt{lbl}, \sigma, \mathtt{stk})(x) &= \sigma(x) \end{aligned}$$

Where $\#_{\mathtt{lbl}}(\mathtt{stk})$ returns the number of occurrences of `lbl` in `stk`.

That is, the marking $\mathtt{marking}(\mathtt{lbl}, \sigma, \mathtt{stk})$ puts a token in the place `lbl` if `lbl` is the return address of a still ongoing call. There are exactly $\#_{\mathtt{lbl}}(\mathtt{stk})$ such calls. It also adds one extra token to `lbl` in order to indicate what is the next label to be executed like a program counter. Last, there are as many token in the variable places as the value of the variable.

Some transitions may have the same name because the instruction appears at many places in the program. This is not a problem. One can easily differentiate them by indexing them (*e.g.* with the corresponding label). However, we try to keep the notation light in order to show that program trace and occurrence sequence play the same role. Hence, it is straightforward to prove that for each program run on some inputs, there is a firing of the SCPN.

[1] If you have colors, you can see that the red part of the net corresponds to the control flow, the green part to variables and, later, a blue part (corresponding to memory management) will be added

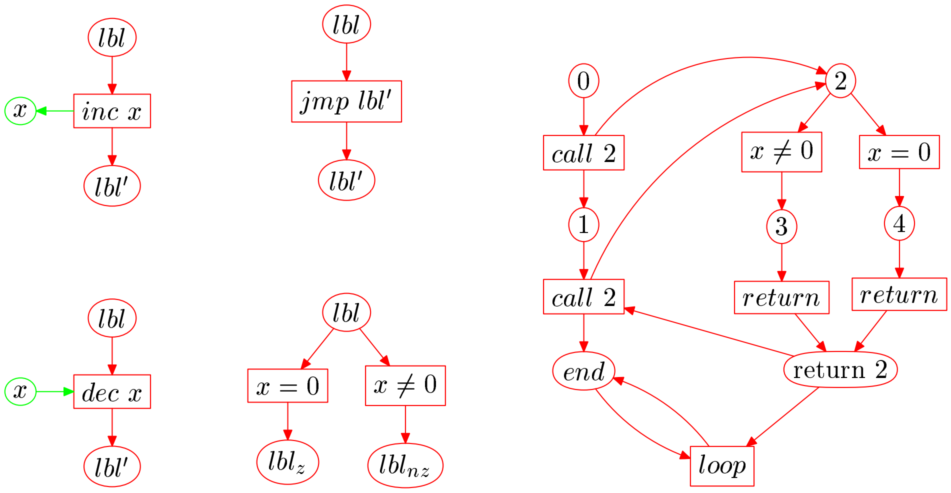

Figure 2: SCPN of instructions.

Figure 3 gives the SCPN of the two version of the addition given before. Figures 4 and 5 give the SCPN of *copy* and *dup*, together with the condensed representation of the macros that will be used later. Note that for the two loops in the *dup* macro, only the net effect of each box on variables has been shown, the specific effect of each transition is more complicated and not depicted here. Figure 6 shows the SCPN of Euclide's algorithm. Note that some places and transitions have been condensed in order to have a smaller net. All of this (including macros) should be expanded.

4.2 Resource Termination

So far, we have seen that a Petri net can be used to simulate program execution in a non-deterministic way with no great difficulty. Now, by looking globally at the number of tokens in variable places, we will be able to decide if a program execution is terminating. This idea is similar to size change termination principle of Lee, Jones and Ben-Amram [5]. They consider every potentially infinite execution and the chains of recursive call generated by such executions. Then, they look at the (infinite) call chain of arguments thus defined and say that the program terminates if each infinite execution enclose at least one chain of always decreasing arguments. However, this criterion doesn't handle precisely what happens to variables and is, for example, unable do decide that the sequence $x = x + 1; x = x - 2$ is terminating, because it is necessary to count carefully resources in this example.

Theorem 14. *Let* pgm *be a program and* $N = SCPN(pgm)$ *be its Size Change Petri Net. Let* M *be a marking such that* $M \trianglerighteq \mathtt{marking}(s)$*. Let* ω *be a program trace from the state* s*. Then* ω *is also an occurrence sequence from the marking* M*.*

Suppose that ω *is a finite program trace, and that* $s \xrightarrow{\omega} s'$*. Then there is a marking* M' *such that* $M \xrightarrow{\omega} M'$ *and* $M' \trianglerighteq \mathtt{marking}(s')$*.*

Of course, there are others firings which do not correspond to a computation, that's why we say that the SCPN simulates a program in a non-deterministic way.

Corollary 15.

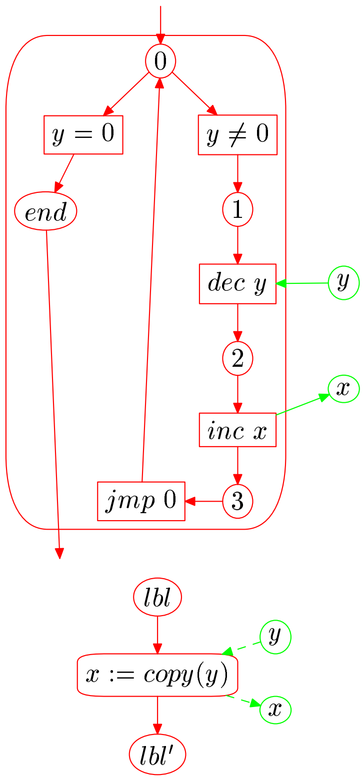

Figure 4: SCPN of *copy* and the macro version.

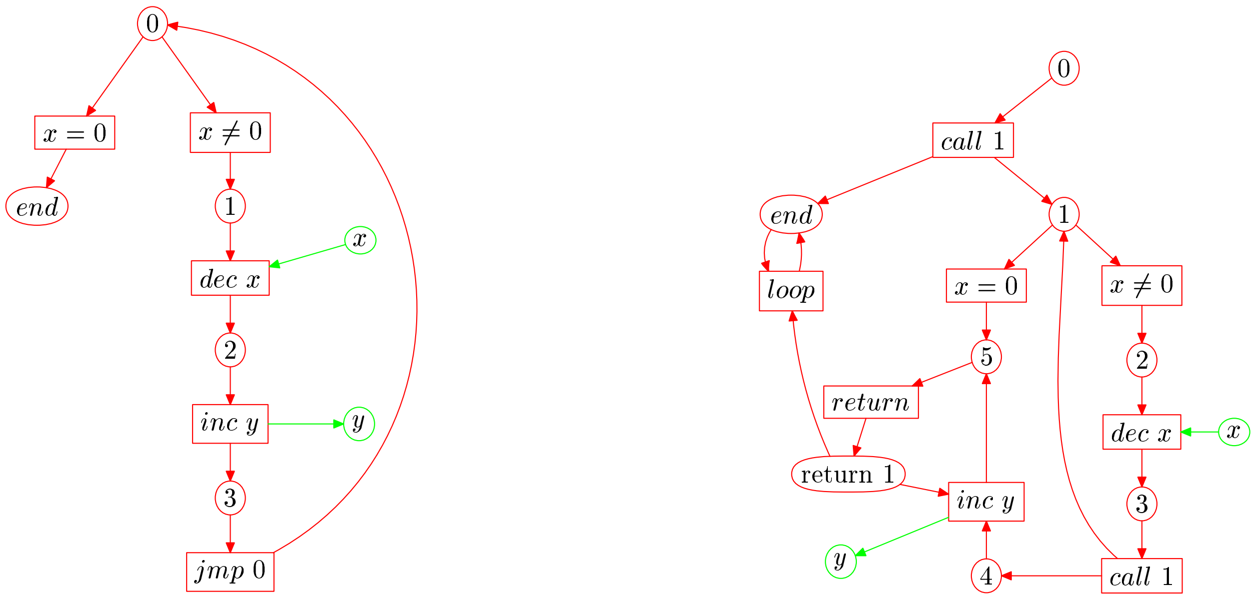

Figure 3: SCPN of both version of Addition, tail-recursive on the left.

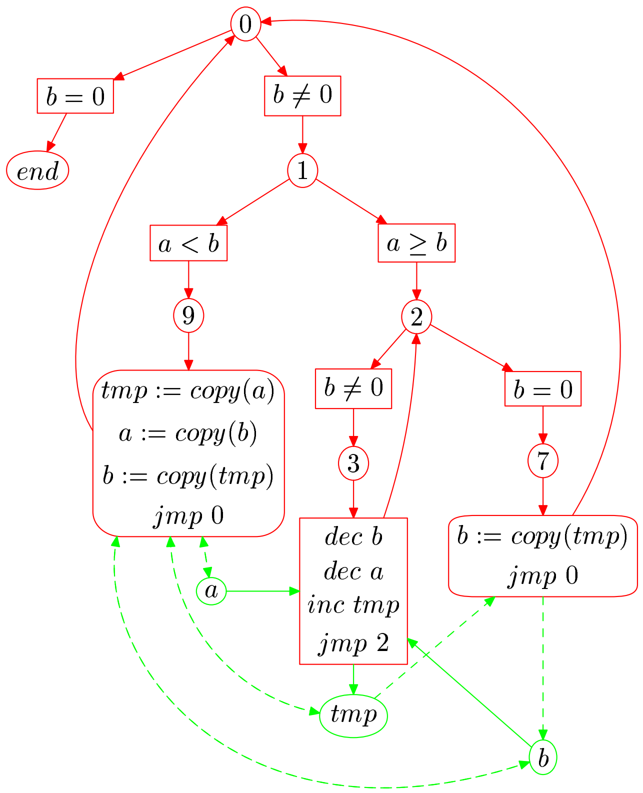

Figure 6: Simplified SCPN of Euclide's algorithm.

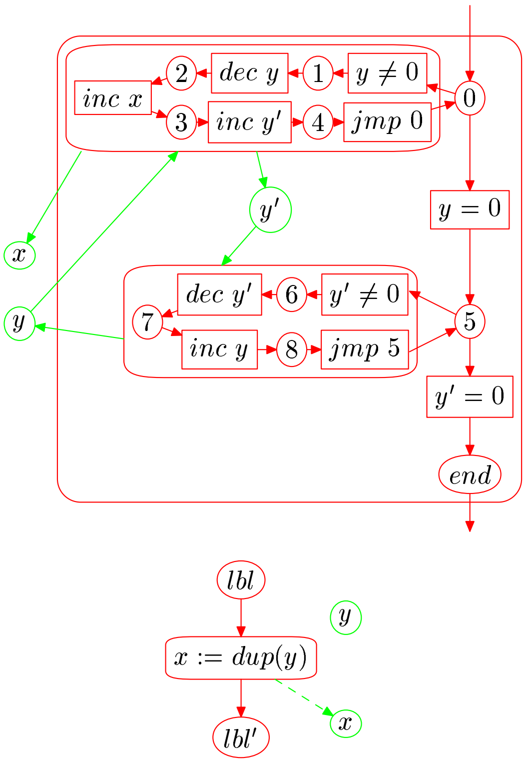

Figure 5: SCPN of dup and the macro version.

- *If the program doesn't terminate on all inputs, then there exists an infinite firing sequence of the Petri net.*
- *Let pgm be a program and $N = SCPN(pgm)$. If N starves* (i.e. *there is no infinite execution), then pgm terminates on all inputs.*

4.3 Detecting an infinite firing sequence

Now, we show that deciding if a Petri net has an infinite firing sequence is equivalent to deciding if a system of linear inequations has a solution. The construction of the system is very similar to the one given by Lien in [6].

Theorem 16. *Deciding if a Petri net has an infinite firing sequence is in NP.*

In order to prove this result, it is simpler to see in a somewhat different way a Petri net $N = (S, T, F)$. As said before, marking are represented by vectors $M \in \mathbf{N}^{|S|}$, given an arbitrary fixed enumeration of the places. A transition is represented by a vector $t \in \mathbf{Z}^{|S|}$ where each component of the vector is the number of token removed (if negative) or added (if positive) in the corresponding place. If M and M' are markings and t is a transition such that $M \xrightarrow{t} M'$, then we have the following vector equation $M + t = M'$.

Definition 17. *The characteristic matrix of a net is the matrix Γ whose columns are the vectors corresponding to each transition given an arbitrary enumeration of the transition.*

The characteristic vector of an occurrence sequence ω is the vector $F(\omega)$ such that the ith component $F(\omega)_i$ is the number of occurrences of the ith transition in ω.

Lemma 18. *Let M and M' be markings and ω be an occurrence sequence such that $M \xrightarrow{\omega} M'$. We have the following matricial equation: $M + \Gamma.F(\omega) = M'$.*

We are now ready to demonstrate the above Theorem.

Proof. Assume that there is an infinite firing sequence. So there is an initial marking M_0 enabling an infinite occurrence sequence ω. Both M_0 and ω define an infinite sequence of markings (M_n) which are the intermediates markings reached during the firing of ω.

There exist i and j such that $i < j$ and $M_i \trianglelefteq M_j$ because the ordering is well-founded (Recall that $M_i \trianglelefteq M_j$ if each component of M_i is less or equal to the corresponding component of M_j).

By construction of M_i and M_j, there exists a finite occurrence sequence ω' such that $M_i \xrightarrow{\omega'} M_j$. So there exists a vector $F(\omega')$ such that $M_i + \Gamma.F(\omega') = M_j \trianglerighteq M_i$. So, $\Gamma.F(\omega') \trianglerighteq 0$.

Conversely, if there exists a vector $X \in \mathbf{N}^{|S|}$ such that $\Gamma.X \trianglerighteq 0$, it is not difficult to build an occurrence sequence ω such that $F(\omega) = X$ and to find a marking M enabling ω. The marking reached after firing ω also enables ω, and so on, so there is an infinite firing sequence of the Petri net.

So, to determine whether there is an infinite firing sequence it suffices to find a integer solution to the following system of linear inequalities:

$$\Gamma.X \geq 0 \tag{1}$$
$$X \geq 0 \tag{2}$$
$$\sum X_i \geq 1 \tag{3}$$

This problem is in NP. $\square$

Theorem 19. *Resource Termination is a co-NP problem.*

The characteristic matrix of the SCPN of addition is the following:

	$x=0$	$x \neq 0$	dec x	incy	jmp 0
0	-1	-1	0	0	1
1	0	1	-1	0	0
2	0	0	1	-1	0
3	0	0	0	1	-1
end	1	0	0	0	0
x	0	0	-1	0	0
y	0	0	0	1	0

The only solution of the system $\Gamma.X \geq 0$ is $X = 0$, so the Petri net has no infinite firing sequence, so the program always terminates.

Similarly, it is easy to build the matrix for the other programs and see that they always terminate.

This Resource Termination principle is quite different from the Size Change Principle of [5]. Our method looks globally at variables and resources while SCP only looks locally at what happens to variables. So, we are able to determine termination of more tricky examples, such as Euclide's algorithm. Moreover, the Size Change Problem is PSPACE complete while our problem is only co-NP. Of course, this implies that there are programs whose termination is decidable with the SCP and not via resource termination. This is mainly due to the lack of expressive power of L, making it really hard to write truly recursive functions such as Ackermann's.

5 Non-size increasing computation

We investigate now non-size increasing (NSI) computation. Roughly speaking no extra memory is required to perform a computation. The maximum, the addition, the gcd are computed by NSI programs. At the beginning of the computation, the memory allocated is given by the size of the inputs. Notice that the size and the value are here the same because we access to data only by the `inc` and `dec` operations, so we have $|v| = v$.

Definition 20. *A program* pgm *is non-size increasing if there is a constant c such that for all inputs $v_1, \cdots, v_m$, if* pgm $\vdash 0, \{i_1 \leftarrow v_1, \ldots, i_n \leftarrow v_m\}, [] \xrightarrow{*} \texttt{lbl}, \sigma, \texttt{stk}$, *we have* $\sum_x |\sigma_i(x)| \leq \sum_{i=1,m} |v_i| + c$.

5.1 Resource Petri Net

A resource Petri net (RPN) of a program `pgm` is similar to a size change Petri net but it has one more place which represents the memory. New transitions of RPN are displayed in Figure 7. The number of tokens in the memory place indicates the free resources. Each time a `dec` instruction is performed, a token is returned to the memory place. This indicates that the program returns a cell which can be reused later in response to a `inc` instruction. Inversely, a `inc` instruction requires a token from the memory place. In other words, a token is reclaimed in order to allocate one cell and so memory place is like the heap. Moreover, we add a guard on `jz` transition in order to avoid to take the branch $x > 0$ when $x = 0$. Otherwise, the first `dec` instruction produces a deadlock.

Definition 21 (initial and final markings). *An initial marking is a marking corresponding to the beginning of the execution of a program, so there is a token in* 0*, some token in the input variables and memory places. A marking M is* initial *if there is a substitution σ_i such that $\sigma_i(x) = 0$ for each non-input variable and $M(p) = \texttt{marking}(0, \sigma_i, [])(p)$ for each non-memory place p.*

A final marking corresponds to the end of the execution of a program. So we have a token in the end place and some in variables places. A marking M is final *if there is a substitution σ such that $M(p) = \texttt{marking}(end, \sigma, [])(p)$ for each non-memory place p.*

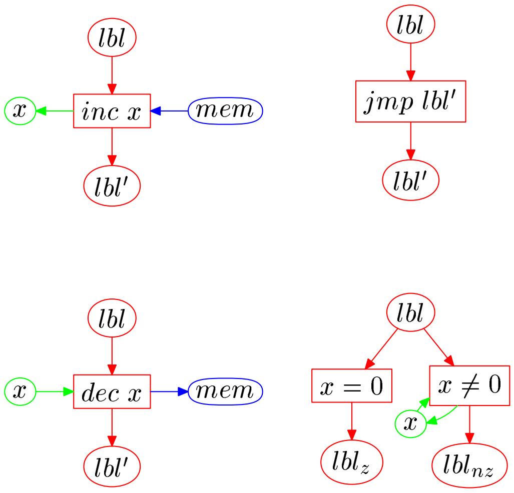

Figure 7: Resource Petri net base transitions

Notice that a final marking is always a deadlock because each transition has a label place (different from end) in its preset

Definition 22. *The resource Petri net of a program* pgm *is* resource-free *if and only if there is a constant c such that each maximal firing sequence starting from an initial marking with c tokens in* mem *leads to a final marking.*

Theorem 23. *The program* pgm *is NSI if the resource Petri net of* pgm *is resource-free.*

Proof. Suppose that the program `pgm` is not NSI. There is a execution of `pgm` on some inputs which requires extra memory. More precisely a `inc` instruction is performed and there is no more token in the memory place. As we have seen previously, each program execution is faithfully simulated by a firing sequence. In this case, we see that the net halts because the `inc` transition is not enabled. ☐

Remark 24. *The converse is unfortunately false. The following program computes nothing really useful. It is NSI but we are not able to detect it. The resource net of this program is shown on Figure 8. The loop between O and 5 violates the condition.*
input: x, y

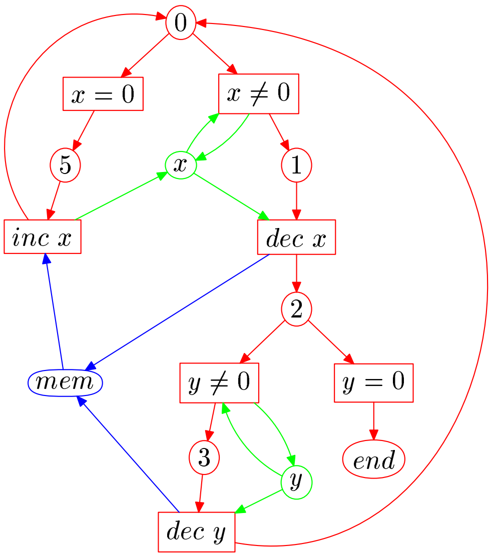

Figure 8: The converse is false

output: y

0	:	$jz\ x\ 5$
1	:	$dec\ x$
2	:	$jz\ y\ end$
3	:	$dec\ y$
4	:	$jmp\ 0$
5	:	$inc\ x$
6	:	$jmp\ 0$

end

Theorem 25. *Let* pgm *be a program,* $S = SCPN(pgm)$ *and* N *be the resource Petri net of* pgm*. If* S *is starving, deciding if* N *is resource-free, is computable in polynomial time in the size of* N *(i.e. the size of the program* pgm*).*

Proof. For this, we construct a directed graph G from the net N as follows. The nodes are the label place. (We forget the variable places and the memory place). There is an edge from the node a to the node b if there is a transition from the place a to the place b in N. The weight between between a and b is -1 if the transition is `inc`, $+1$ if the transition is `dec`, and 0 otherwise.

A loop is a path starting and ending in the same node a. We compute the minimal weight of each path between any two nodes, including loop. We perform this by computing the transitive closure of the adjacency matrix of G. The runtime is in $O(n^4)$ where n is size of G. The Petri net N is resource-free if each loop has a positive or null weight. This condition ensures that we have always enough memory to fire it in N. (Indeed, assume we have detected a loop of negative weight. By firing this loop sufficiently enough, we will run out of memory and so N will reach a deadlock which not a final marking.) The initial amount of memory is given by the minimal weight of a path. In conclusion, each maximal firing sequence is not halted by lack of memory and is finite and so it necessarily terminates in a final marking. □

6 Future works

1. We have played with integers in order to illustrate our ideas. However, we can extend to data structure like lists, trees . . . In the Appendix, we give the example of the insertion sort and the quicksort.

2. The ressource analysis is based on the memory. In similar way, we can add a stack label in the ressource Petri net in order to control the stack growth rate.

3. The most challenging point is to be able to deal with direct memory access and indirect fetch and store.

4. The Petri net analysis also provides a covering tree of a program execution which is very similar to control flow graph. We have in mind to exploit this to make program transformation.

7 Acknowledgments

The authors would like to thank Bruno Gaujal for his patient explanations about Petri nets and Bernard Helmstetter for his useful hints about the detection of starvation.

References

[1] Desel, J., and Esparza, J. *Free Choice Petri nets.* Cambridge University Press, 1995.

[2] Esparza, J. Decidability and complexity of petri net problems - an introduction. *LNCS 1491* (1998), 374–428.

[3] Hofmann, M. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)* (1999), pp. 464–473.

[4] Hofmann, M. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00* (2000), vol. 1782 of *Lecture Notes in Computer Science*, pp. 165–179.

[5] Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *Symposium on Principles of Programming Languages* (january 2001), vol. 28, ACM press, pp. 81–92.

[6] Lien, Y. Termination properties of generalized petri nets. *SIAM journal of computing 5*, 2 (1976), 251–265.

[7] Shepherdson, J., and Sturgis, H. Computability of recursive functions. *Journal of the ACM 10*, 2 (1963), 217–255.

A Some more examples

A.1 Multiplication

The following program computes the multiplication of two numbers:
input variables: u, v.

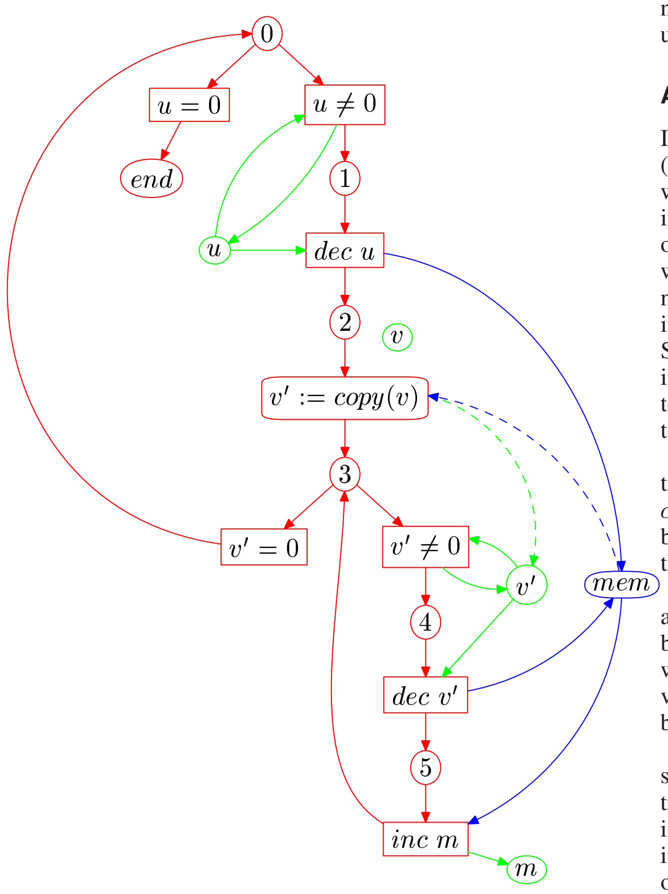

Figure 9: RPN of multiplication

output variable: m.

$$\begin{array}{rcl} 0 & : & jz\ u\ end \\ 1 & : & dec\ u \\ 2 & : & v' := v \\ 3 & : & jz\ v'\ 0 \\ 4 & : & dec\ v' \\ 5 & : & inc\ m \\ 6 & : & jmp\ 3 \\ end & & \end{array}$$

The RPN of the program is shown on Figure 9. Note that the net has been condensed in some points.

The system $\Gamma.X \geq 0$ has no strictly positive solution, so the program always terminates. Again, this is not due to a lack of resources.

The covering graph has not been build here, it is very easy to see that the presence of $loop_1$ leads quickly to a node $(2, -\infty)$ in the graph.

All execution stop because the second loop cannot be fired infinitely many time without emptying v' and v' can only be fulled by firing the first loop and that cannot be done infinitely often without emptying u. So the program always terminates.

The program is not NSI because the dup macro can be fire many time and inside it, even if v is not empty, the first box of the dup macro may cause a deadlock if mem is empty. The number of extra tokens needed is not a constant and depends upon v.

A.2 insertion sort

In order to compute a sorting algorithm, we have to use lists (or arrays) of integers. Lists are not allowed in L. The proper way to do this would be to introduce a coding of lists into integers and use this coding, but this would lead to a quite obfuscated program. So we choose to extend the language with lists. In sorting algorithms, the elements of the lists have no computational meaning. Thus we can totally forgive them in the analysis (they play no role in termination). So, in the SCPN, there will be places only for lists variables and not for integers (and memory management will only be counted in terms of length of the lists, that is all integers are considered to use the same amount of memory).

In order to build and destroy lists, we had two new instructions, $cons$ and hd, similar to the inc and dec on integers. $cons(a, l)$ take an integer and a list and had the integer at the beginning of the list. $a = hd(l)$ remove the first element of the list and return its value in a.

Similarly, the test jz is modified in order to work on lists and will now jump only if the list is empty. Last, we add a build-in comparison function on integer $if\ a < b\ jmp$ `lbl` which jumps if and only if the value of a is smaller than the value of b. Again, we may put a build-in function here only because the integers play no role in termination.

The following program inserts an integer in an already sorted list. Note that we need to have an accumulator to keep trace of the beginning of the list and put it back in place after insertion has been performed.
input variables: a, l.
output variable: l.

$$\begin{array}{rcl} I0 & : & jz\ l\ I6 \\ I1 & : & b := hd(l) \\ I2 & : & if\ a \leq b\ jmp\ I5 \\ I3 & : & cons(b, acc) \\ I4 & : & jmp\ I0 \\ I5 & : & cons(b, l) \\ I6 & : & cons(a, l) \\ I7 & : & jz\ acc\ Iend \\ I8 & : & b := hd(acc) \\ I9 & : & cons(b, l) \\ I10 & : & jmp\ I7 \\ Iend & & \end{array}$$

The RPN of $insert$ is shown on the left part (undashed) of Figure 10. It always terminates because neither the first loop $(I0, I1, I2, I3)$ nor the second one $(I7, I8, I9)$ can run forever without emptying l or acc. It is NSI if there is one token in mem at the beginning of the execution.

The following program computes the insertion sort, using the insertion has defined previously. The label $Iend$ of the insertion is merged with the label 0 of the sort. In order to keep the two nets distinct on the Figure, we keep the labels distinct and put an arrow between them.
input variable: l'.
output variable: l.

$$\begin{array}{rcl} 0 & : & jz\ l'\ end \\ 1 & : & a := hd(l') \\ 2 & : & jmp\ I0 \\ end & & \end{array}$$

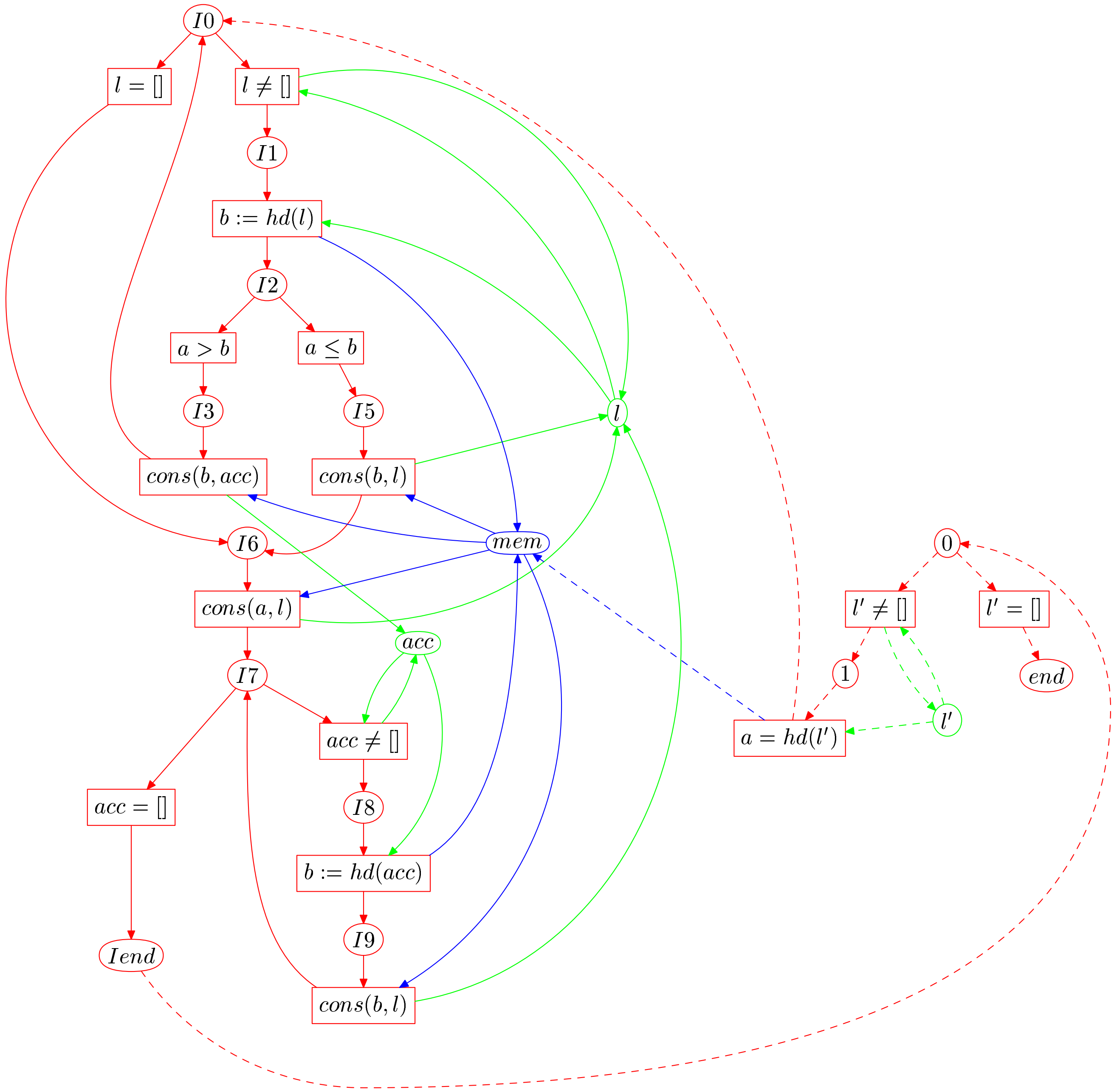

Figure 10: RPN of insertion sort

The RPN of the insertion sort is shown on Figure 10. It always terminates because the loop $(0, 1, I0, \ldots, Iend)$ cannot run forever without emptying l'. It is NSI and don't need any token in mem at the beginning of the execution.

It is quite easy to build the characteristic matrix and see that the RPN is resource-free and see that the program does terminate on all inputs and is non size increasing.

A.3 Quicksort

This section shows that this analysis apply well on the quicksort (and so potentially to all "divide and conquer" algorithms), a program which is rather tricky and hard to deal with usually.

In order to deal with the recursives call and the stack of values, we'll have to introduce another trick. $cons$' now have 3 arguments. The second and third are the usual ones. The first one is an integer, counting a number of **tag**s on the list. If a list starts with a tagged $cons$, then it is considered empty for all purpose (especially for jz instructions). The function tag (resp. $untag$) add (resp. remove) 1 from the tag argument of the first $cons$ of the list, and does nothing on nil. The function hd only works if the first element is not tagged (*i.e.* tag value of 0) and the function $cons$ create a list with no tag on its first element. Size of list is now twice as big as without tags, but we will still count a size of 1 for each $cons$, knowing that it is actually a size of 2 (as this is the only size measurement we use here, this has no consequence on memory management). Thus, tag and $untag$ will not alter the size of the lists.

A.3.1 concat

Let us first start with the concatenation of 2 lists, adding an extra element in the middle.
input variables: l_1, a, l_2
output variables: l'

$$\begin{array}{rcl} C0 & : & jz\ l_1\ C4 \\ C1 & : & b = hd(l_1) \\ C2 & : & cons(b, acc') \\ C3 & : & jmp\ C0 \\ C4 & : & cons(a, acc') \\ C5 & : & jz\ l_2\ C9 \\ C6 & : & b = hd(l_2) \\ C7 & : & cons(b, acc') \\ C8 & : & jmp\ C5 \\ C9 & : & jz\ acc'\ Cend \\ C10 & : & b = hd(acc') \\ C11 & : & cons(b, l') \\ C12 & : & jmp\ C9 \\ Cend & & \end{array}$$

Note that the lack of recursive call stack forces us to use and abuse of temporary accumulators (as done for the insertion sort).

It is very easy to see that the program always terminates and is non size increasing if it has at least one token in mem.

A.3.2 split

Then, the function that takes a list and splits it, according to another element.
input variables: a, l
output variables: l_1, l_2

$$\begin{array}{rcl} S0 & : & jz\ l\ Send \\ S1 & : & b = hd(l) \\ S2 & : & if\ a < b\ then\ jmp\ S5 \\ S3 & : & cons(b, l_1) \\ S4 & : & jmp\ S0 \\ S5 & : & cons(b, l_2) \\ S6 & : & jmp\ S0 \\ Send & & \end{array}$$

Again, it is very easy to see that the program always terminates and is non size increasing.

A.3.3 sort

Last but not least, comes the sorting algorithm itself. Note how tagging and untagging is used to keep track of older results (or thing yet to be calculated) inside variables that are still in use. Note also that the 4 uses of the $copy$ macro will lead in practice to 4 different piece of code as the copy is not done between the same variables each time. The full program, with all the macros flatten and the subroutines put together as 70 instructions. Such a big number is easily explained by the lack of expressive power of the language.

The calls to the $concat$ and $split$ subroutines have been done by means of jmp because they're unique. So we don't need the full power of the $call/return$ pair and can avoid its complexity.
input variable: l
output variable: l'

$$\begin{array}{rcl} 0 & : & jz\ l\ 16 \\ 1 & : & a = hd(l) \\ 2 & : & cons(a, acc) \\ 3 & : & jz\ l\ 14 \\ 4 & : & jmp\ S0 \\ 5 = Send & : & tag\ l_2 \\ 6 & : & l := copy(l_1) \\ 7 & : & call\ 0 \\ 8 & : & l_1 := copy(l') \\ 9 & : & tag\ l_1\ ;\ untag\ l_2 \\ 10 & : & l := copy(l_2) \\ 11 & : & call\ 0 \\ 12 & : & l_2 := copy(l') \\ 13 & : & untag\ l_1 \\ 14 & : & a = hd(acc) \\ 15 & : & jmp\ C0 \\ 16 & : & return \\ end & & \end{array}$$

Again, the analysis shows that quicksort always terminates and is non size increasing. Not that the program actually need some extra memory to store the return addresses (*i.e.* the return stack). If one want to detect that, if can easily be done by adding arrows from mem to `calls` and from $returns$ to mem. With this meaning, the quicksort is not non size increasing.