Static analysis of cloud elasticity[*]

Abel Garcia Cosimo Laneve Michael Lienhardt

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus

Abstract

We propose a *static analysis technique* that computes upper bounds of virtual machine usages in a *concurrent* language with explicit acquire and release operations of virtual machines. In our language it is possible to delegate other (ad-hoc or third party) codes to release virtual machines (by passing them as arguments of invocations). Our technique is modular and consists of (*i*) a type system associating programs with behavioural types that records relevant informations for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioral types and return cost equations, and (*iii*) an automatic solver for the the cost equations.

We have experimentally evaluated our technique using a cost analysis solver and we report some results. The technique in this paper may be also applied to estimate (heap) memory consumptions in object-oriented languages.

1 Introduction

The analysis of resource usage in a program is of great interest because an accurate assessment could reduce energy consumption and allocation costs. These two criteria are even more important today, in modern architectures like mobile devices or cloud computing, where resources, such as virtual machines, have hourly or monthly rates. In facts, cloud computing introduces the concept of *elasticity*, namely the possibility for virtual machines to scale according to the software needs. In order to support elasticity, cloud providers, including Amazon, Google, and Microsoft Azure, (1) have pricing models that allow one to hire on demand virtual machine instances and paying them for the time they are in use, and (2) have APIs that include instructions for requesting and releasing virtual machine instances.

While it is relatively easy to estimate worst-case costs for sample codes, extrapolating this information for fully real-life complex programs could be

[*]Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

cumbersome and highly error-sensitive. The first attempts about the analysis of resource usages dates back to Wegbreit's pioneering work in 1975 [20], which develops a technique for deriving closed-form expressions out of programs. The evaluation of these expressions would return upper-bound costs that are parametrised by programs' inputs.

Wegbreit's contribution has two limitations: it addresses a simple functional languages and it does not formalize the connection between the language and the closed-form expressions. A number of techniques have been developed afterwards to cope with more expressive languages (see for instance [3, 9]) and to make the connection between programs and closed-form expressions precise (see for instance [19, 13]). We postpone the discussion of the related work in the literature in Section 8.

To the best of our knowledge, current cost analysis techniques always address (concurrent) languages featuring only addition of resources. When removal of resources is considered, it is used in a very constrained way [4]. On the other hand, cloud computing elasticity requests powerful acquire operations *as well as* release ones. Let us consider the following problem: given a pool of virtual machine instances and a program that acquires and releases these instances, what is the minimal cardinality of the pool guaranteeing the execution of the program without interruptions caused by lack of virtual machines? Under the assumption that one can acquire a virtual machine that has been previously released. A solution to this problem is useful both for cloud providers and for cloud customers. For the formers, it represents the possibility to estimate *in advance* the resources to allocate to a specific service. For the latter ones, it represents the possibility to pay *exactly* for the resources that are needed.

It is worth to notice that, without a full-fledged release operation, the cost of a concurrent program may be modeled by simply aggregating the sets of operations that can occur in parallel, as in [5]. By full-fledged release operation we mean that it is possible to delegate other (ad-hoc or third party) methods to release resources (by passing them as arguments of invocations). For example, consider the following method

```
Int double_release(VM x, VM y) {
  release x; release y;
  return 0 ;
}
```

that takes two machines and simply releases them. The cost of this method depends on the machines in input:

– it may be `-2` when `x` and `y` are *different* and active;

– it may be `-1` when `x` and `y` are *equal* and active – consider the invocation `double_release(x,x)`;

– it may be `0` when the two machines have been already released.

In this case, one might over-approximate the cost of `double_release` to `0`. However this leads to disregard releases and makes the analysis (too) imprecise.

In order to compute the cost of methods like `double_release` in a precise way, in Section 4 we associate methods with abstract descriptions that also carry informations about parameter states and their identities. These descriptions are called *behavioural types* and are formally connected to the programs by means of a typing system.

In Section 5 we therefore analyse behavioural type descriptions by translating them in codes that are adequate for a powerful off-the-shelf solver – the `CoFloCo` solver [10]. As discussed in [7], in order to compute tight upper bounds, we have two functions per method: a function computing the *peak cost* – *i.e.* the worst case cost for the method to complete – and a function computing the *net cost* – *i.e.* the cost of the method after its completion. In facts, the functions that we associate to a method are much more than two. The point is that, if a method has two arguments – see `double_release` – and it is invoked with the two arguments equal then its cost cannot be computed by function taking two arguments, but it must be computed by a function with one argument only. This means that, for every method and *every partition of its arguments*, we define two cost functions: one for the peak cost and the other for the net cost.

In Section 7 we have we report the results of some of our experimental evaluation. In particular, we compute the cost of `double_release` and two implementation of the factorial functions by means of `CoFloCo`.

Our technique target a simple concurrent language with explicit operations of creation and release of resources. The language is defined in Section 2 and we discuss restriction that ease the development of our technique in Section 3. We discuss how these restriction can be removed and outline our correctness proof in Section 6. We deliver concluding remarks in Section 9.

In this paper we use the metaphor of cloud computing and virtual machines. We observe that our technique actually addresses every type of resources that retain operations of acquire (or creation) and release, such as heap usages in concurrent object-oriented languages.

2 The language `vml`

The syntax and the semantics of `vml` are defined in the following two subsections; the third subsection discusses a number of examples.

Syntax. A `vml` program is a sequence of method definitions $T\,\mathtt{m}(\overline{T\ x})\{\,\overline{F\ y\ ;}\ s\,\}$, ranged over by M, plus a main body $\{\overline{F\,z\ ;}\ s'\}$. In `vml` we distinguish between *simple types* T which are either integers `Int` or virtual machines `Vm`, and *types* F, which also include *future types* `Fut<Int>`. These future types let asynchronous method invocations be typed (see below). The notation $\overline{T\ x}$ denotes any finite sequence of *variable declaration* $T\ x$. The elements of the sequence are separated by commas. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1\ ;\ \cdots\ ;\ T_n\ x_n\ ;$ when the sequence is not empty; we mean the empty sequence otherwise.

The syntax of statements s, expressions with side-effects z and expressions e of `vml` is defined by the following grammar:

$$
\begin{array}{rcl}
s & ::= & x = z \;\mid\; \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s \,\} \;\mid\; \texttt{return}\ e \;\mid\; s\ \texttt{;}\ s \;\mid\; \texttt{release}(e) \\
z & ::= & e \;\mid\; e\texttt{!m}(\overline{e}) \;\mid\; e\texttt{.get} \;\mid\; \texttt{new vm} \\
e & ::= & \texttt{this} \;\mid\; se \;\mid\; nse
\end{array}
$$

A (*pure*) expression e is either an integer constant p, or a variable x, or the reserved identifier `this`, or the standard arithmetic, relational and boolean operations. Since our analysis will be parametric with respect to the inputs, we will parse expressions in a careful way. In particular we split them into *size expressions* se, which are expressions in presburger arithmetics, and *non-size expressions* nse, which are the other type of expressions. The syntax of size and non-size expressions is the following:

$$
\begin{array}{rcl}
nse & ::= & p \;\mid\; x \;\mid\; nse \leq nse \;\mid\; nse\ \texttt{and}\ nse \;\mid\; nse\ \texttt{or}\ nse \;\mid\; nse + nse \;\mid\; nse - nse \\
 & \mid & nse \times nse \;\mid\; nse / nse \\
se & ::= & ve \;\mid\; ve \leq ve \;\mid\; se\ \texttt{and}\ se \;\mid\; se\ \texttt{or}\ se \\
ve & ::= & p \;\mid\; x \;\mid\; ve + ve \;\mid\; p \times ve \\
p & ::= & \textit{integer constants}
\end{array}
$$

An expression z may change the state of the system. In particular, it may be an *asynchronous* method invocation that does not suspend caller's execution: when the value computed by the invocation is needed then the caller performs a `get` operation. Expressions z also include `new vm` that creates a new virtual machine. Operations taking place on different virtual machines may execute in parallel, while operations in the same virtual machine interleave their evaluation.

A statement s may be either one of the standard operations of an imperative language or the `release` operation. The operation `release`(x) marks the virtual machine x for disposal. Method invocations performed by a released machine, as well as, creations and releases of machines, always return erroneous values.

In the whole paper, we assume that sequences of declarations $\overline{T\ x}$ and method declarations $\overline{M}$ do not contain duplicate names.

Semantics. `vml` semantics is defined as a transition relation between *configurations*, noted cn and defined below

$$
\begin{array}{rcl}
cn & ::= & \epsilon \;\mid\; \mathit{fut}(f, v) \;\mid\; \mathit{vm}(o, a, p, q) \;\mid\; \mathit{invoc}(o, f, \texttt{m}, \overline{v}) \;\mid\; cn\ cn \\
p & ::= & \{ l \mid s \} \;\mid\; \texttt{idle} \\
q & ::= & \epsilon \;\mid\; \{ l \mid s \} \;\mid\; q\ q \\
v & ::= & \textit{integer constants} \;\mid\; o \;\mid\; f \;\mid\; \bot \;\mid\; \top \;\mid\; \mathit{err} \\
l & ::= & [\cdots, x \mapsto v, \cdots]
\end{array}
$$

Configurations are sets of elements – therefore we identify configurations that are equal up-to associativity and commutativity – and are denoted by the juxtaposition of the elements $cn\ cn$; the empty configuration is denoted by ε. The transition relation uses two infinite sets of names: *vm names*, ranged over by o, o', $\cdots$ and *future names*, ranged over by f, f', $\cdots$. We assume there are infinitely many vm names and future names. The function fresh() returns either a fresh vm name or a fresh future name; the context will disambiguate between the twos. We also use l to range over maps from variables to values. The map l also binds the special name destiny to a future value.

Runtime values v are either integers or vm and future names, or two distinct special values denoting a machine alive ($\top$) or dead ($\bot$), or an erroneous value err.

The elements of configurations are

- *virtual machines* $vm(o, a, p, q)$ where o is a vm name; a is either $\top$ or $\bot$ according to the machine is alive or dead, p is either $\{l \mid \varepsilon\}$, representing a terminated statement, or is the *active process* $\{l \mid s\}$, where l returns the values of local variables and s is the continuation; q is a set of processes to evaluate.
- *future binders* $fut(f, v)$. When the value v is $\bot$ then the value of f has still to be computed.
- *method invocations* $invoc(o, f, \mathtt{m}, \overline{v})$.

The following auxiliary functions are used in the semantic rules (we assume a fixed `vml` program):

- $\mathrm{dom}(l)$ returns the domain of l.
- $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$.
- $[\![e]\!]_l$ returns the value of e, possibly retrieving the values of the variables that are stored either in l. Operations in `vml` are also defined on the value err: when one of the arguments is err, every operation returns err. $[\![\overline{e}]\!]_l$ returns the tuple of values of $\overline{e}$. When e is a future name, the function $[\![\cdot]\!]_l$ is the identity. Namely $[\![f]\!]_l = f$.
- $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}) = \{[\mathrm{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] \mid s\{^{o}/_{\mathtt{this}}\}\}$, where $T\ \mathtt{m}(\overline{T\ x})\{\overline{T'\ z}; s\}$ belongs to the program.

The transition relation rules are collected in Figure 1. They define transitions of virtual machines $vm(o, a, p, q)$ according to the shape of the statement in p. We focus on rules concerning the method invocations and the management of virtual machines in `vml`, since the other ones are standard.

(New-VM) creates a virtual machine and makes it active. If the virtual machine executing `new vm` has been already released, then the operation returns an error – rule (New-VM-Err). A virtual machine is disposed by means of the operation `release`(x): this amounts to update its state a to $\bot$.

Rule (Async-Call) defines asynchronous method invocation $x = e!\mathtt{m}(\overline{e})$. This rule creates a fresh future name that is assigned to the identifier x. The evaluation of the called method is then transferred to the callee virtual machine – rule (Bind-Mtd) – and the caller progresses without waiting for callee's termination. If the caller has been already disposed then the invocation returns err – rule (Async-Call-Err) The invocation binds err to the future name when either the caller has been released – rule (Async-Call-Err) – or the callee machine has been

$$\frac{\begin{array}{c}\text{(Assign)}\\ x \in \mathrm{dom}(l) \quad v = \llbracket e \rrbracket_l\end{array}}{\begin{array}{c} vm(o,a,\{l \mid x = e; s\}, q) \\ \rightarrow vm(o,a,\{l[x \mapsto v] \mid s\}, q)\end{array}} \qquad \frac{\begin{array}{c}\text{(Read-Fut)}\\ f = \llbracket e \rrbracket_l \quad v \neq \bot\end{array}}{\begin{array}{c} vm(o,a,\{l \mid x = e.\mathtt{get}; s\}, q)\ \mathit{fut}(f,v) \\ \rightarrow vm(o,a,\{l \mid x = v; s\}, q)\ \mathit{fut}(f,v)\end{array}}$$

$$\frac{\begin{array}{c}\text{(New-VM)}\\ o' = \mathrm{fresh}(\mathtt{VM})\end{array}}{vm(o,\top,\{l \mid x = \mathtt{new\ vm}; s\}, q) \rightarrow vm(o,\top,\{l \mid x = o'; s\}, q)\ vm(o',\top,\{\varnothing | \varepsilon\}, \varnothing)}$$

$$\frac{\begin{array}{c}\text{(Async-Call)}\\ o' = \llbracket e \rrbracket_l \quad \overline{v} = \llbracket \overline{e} \rrbracket_l \quad f = \mathrm{fresh}(\ )\end{array}}{\begin{array}{c} vm(o,\top,\{l \mid x = e!\mathtt{m}(\overline{e}); s\}, q) \\ \rightarrow vm(o,\top,\{l \mid x = f; s\}, q)\ \mathit{invoc}(o', f, \mathtt{m}, \overline{v})\ \mathit{fut}(f,\bot)\end{array}} \qquad \frac{\begin{array}{c}\text{(Bind-Mtd)}\\ \{l \mid s\} = \mathrm{bind}(o, f, \mathtt{m}, \overline{v})\end{array}}{\begin{array}{c} vm(o,\top,p,q)\ \mathit{invoc}(o, f, \mathtt{m}, \overline{v}) \\ \rightarrow vm(o,\top,p,q \cup \{l \mid s\})\end{array}}$$

$$\frac{\begin{array}{c}\text{(Release-VM)}\\ o' = \llbracket e \rrbracket_l \quad o \neq o'\end{array}}{\begin{array}{c} vm(o,\top,\{l \mid \mathtt{release}(e); s\}, q)\ vm(o',a',p',q') \\ \rightarrow vm(o,\top,\{l \mid s\}, q) \quad vm(o',\bot,p',q')\end{array}} \qquad \frac{\begin{array}{c}\text{(Release-VM-self)}\\ o = \llbracket e \rrbracket_l\end{array}}{\begin{array}{c} vm(o,a,\{l \mid \mathtt{release}(e); s\}, q) \\ \rightarrow vm(o,\bot,\{l \mid s\}, q)\end{array}}$$

$$\frac{\begin{array}{c}\text{(Cond-True)}\\ \llbracket e \rrbracket_l \neq 0\end{array}}{\begin{array}{c} vm(o,a,\{l \mid \mathtt{if}\ e\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_2\}; s\}, q) \\ \rightarrow vm(o,a,\{l \mid s_1; s\}, q)\end{array}} \qquad \frac{\begin{array}{c}\text{(Cond-False)}\\ \llbracket e \rrbracket_l = 0 \quad \text{or} \quad \llbracket e \rrbracket_l = \mathit{err}\end{array}}{\begin{array}{c} vm(o,a,\{l \mid \mathtt{if}\ e\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_2\}; s\}, q) \\ \rightarrow vm(o,a,\{l \mid s_2; s\}, q)\end{array}}$$

$$\begin{array}{c}\text{(Activate)}\\ vm(o,a,\{l' \mid \varepsilon\}, q \cup \{l \mid s\}) \\ \rightarrow vm(o,a,\{l \mid s\}, q)\end{array} \qquad \frac{\begin{array}{c}\text{(Return)}\\ v = \llbracket e \rrbracket_l \quad f = l(\mathrm{destiny})\end{array}}{\begin{array}{c} vm(o,a,\{l \mid \mathtt{return}\ e; s\}, q)\ \mathit{fut}(f,\bot) \\ \rightarrow vm(o,a,\{l \mid s\}, q)\ \mathit{fut}(f,v)\end{array}} \qquad \frac{\begin{array}{c}\text{(Context)}\\ cn \rightarrow cn'\end{array}}{cn\ cn'' \rightarrow cn'\ cn''}$$

$$\begin{array}{c}\text{(New-VM-Err)}\\ vm(o,\bot,\{l \mid x = \mathtt{new\ vm}; s\}, q) \\ \rightarrow vm(o,\bot,\{l[x \mapsto \mathit{err}]; s\}, q)\end{array} \qquad \frac{\begin{array}{c}\text{(Async-Call-Err)}\\ f = \mathrm{fresh}(\ )\end{array}}{\begin{array}{c} vm(o,\bot,\{l \mid x = e!\mathtt{m}(\overline{e}); s\}, q) \\ \rightarrow vm(o,\bot,\{l \mid x = f; s\}, q)\ \mathit{fut}(f,\mathit{err})\end{array}}$$

$$\begin{array}{c}\text{(Bind-Mtd-Err)}\\ vm(o,\bot,p,q)\ \mathit{invoc}(o, f, \mathtt{m}, \overline{v})\ \mathit{fut}(f,\bot) \\ \rightarrow vm(o,\bot,p,q)\ \mathit{fut}(f,\mathit{err})\end{array} \qquad \begin{array}{c}\text{(Bind-Partial)}\\ \mathit{invoc}(\mathit{err}, f, \mathtt{m}, \overline{v})\ \mathit{fut}(f,\bot) \rightarrow \mathit{fut}(f,\mathit{err})\end{array}$$

$$\begin{array}{c}\text{(Release-Bot)}\\ vm(o,\bot,\{l \mid \mathtt{release}(e); s\}, q) \rightarrow vm(o,\bot,\{l \mid s\}, q)\end{array}$$

Figure 1: Semantics of `vml`.

disposed – rule (Bind-Mtd-Err). Rule (Read-Fut) allows the caller to retrieve the value returned by the callee.

The initial configuration of a `vml` program with main function $\{\overline{F\ x\ ;}\ s\}$ is

$$ob(\mathit{start}, \top, \{[\mathrm{destiny} \mapsto f_{\mathit{start}}] \mid s\}, \varnothing)$$

where *start* is a special virtual machine and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$.

Examples. In order to illustrate the features of `vml` we discuss few examples. For every example we also examine the type of output we expect from our cost analysis. We begin with two methods computing the factorial function:

```
Int fact(Int n){
   Fut<Int> x ; Int m ; VM z ;
   if (n==0) { return 1 ; }
   else { z = new VM ; x = z!fact(n-1) ; m = x.get ; release z ; return n*m ;
   }
}
Int cheap_fact(Int n, Int r){
   Fut<Int> x, y ; Int m ; VM z ;
   if (n==0) { return r ; }
   else { z = new VM; x = z!prod(n,r) ; m = x.get ; release z ;
          y = this!cheap_fact(n-1,m) ; m = y.get ; return m ;
   }
}
```

(`prod(x,y)` has been omitted: it just returns `x*y`). The method `fact` is the standard definition of factorial with the recursive invocation `fact(n-1)` performed on a new virtual machine `z`. The caller waits for its result, let it be `m`, then it releases the machine `z` and delivers the value `n*m`. Notice that every vm creation occurs before any release operation. As a consequence, `fact` will create as many virtual machines as the argument n. Therefore, in order to be executed, `fact` needs `n` virtual machines (in addition to the one where the method is performed).

While `cheap_fact` also computes the factorial, its behaviour is different. In particular it implements the function

$$f(n,r) \; = \; \begin{cases} r & \text{if } n=0 \\ f(n-1, n*r) & \text{otherwise} \end{cases}$$

which is initially invoked with $f(n,1)$. In `cheap_fact` the computation of $n*r$ is performed on a new virtual machine, which is released *before* the recursive invocation. For this reason, one might always *reuse* the same virtual machine (from a pool). In facts, it turns out that the cost of `cheap_fact` is `1` (in addition to the virtual machine where the method is performed).

The analysis of the two implementations of factorial has been easy because the `release` operation always carries a locally created virtual machine. Yet, in `vml`, `release` may also apply to method arguments and this is the major source of difficulties for the analysis. Consider for instance the following code:

```
Int first_method() {
  VM x ; Fut<Int> f ; Fut<Int> g ;
  x = new vm;
  f = x!unknown_method(this) ; f.get ;
  g = x!second_method() ; g.get ;
  release x ;
  return 0 ; }
```

A rough analysis might indicate that `first_method` creates a virtual machine, invokes two methods on that machine, and then releases it. However, this analysis is wrong when `unknown_method` releases its argument(s). For instance, if `unknown_method` releases the argument `this`, the invocation of `second_method` and the statement `release x` will not be executed. In order to let the cost analysis be compositional, we record the effects of methods on virtual machines in the arguments and we compute the cost analysis accordingly. One might argue that compositionality might be achieved, in this case, by admitting an over- or under-approximate output instead of recording method's effects on arguments. Actually these approximations are not reasonable if one does not consider releases on the arguments:

– an over-approximation might return very imprecise results. Consider, for instance the case when the invocation `x!unknown_method(this)` releases the two arguments `x` and `this`. The over-approximation will compute the cost of `second_method`, even if it will never be called. This means that the analysis might output a very large cost because second_method creates a large number of virtual machines.

– an under-approximation might return erroneous results, as it will consider that `release x` will be executed (corresponding to a `-1` cost) while we actually have a null cost.

3 Problems, solutions and restrictions

In this section, we present the two new important concepts linked to resource removal, their properties, and which restrictions we put on input programs to keep our analysis from being too complex (indeed, most of the restriction presented here can be relaxed by increasing the complexity of our analysis; this will be discussed in Section 9).

Method's effects. As discussed in the foregoing example `first_method`, in order to augment the precision of the cost analysis and to avoid erroneous outputs, our analysis records the effects that a method has on its interface. To keep the formalism as simple as possible, we restrict these effects to be *a set of virtual machines in method's interface*, which is noted `R` in Section 4. This simplicity – `R` is a set – has a price: the virtual machines in the interface of a method that will be released in *every execution path* is always the same. That is, a method as

```
Int ugly_release(VM x1, ... , VM xn) {
  VM x ; Fut<Int> f ;
  x = new vm ; f = x!release_all(x1, ..., xn) ; release x ; f.get ;
  return 0; }
```

cannot be handled by our analysis (`release_all(`x_1`, ..., `x_n`)` disposes the virtual machines x_1, ..., x_n). In facts, since `release x` is performed before the synchronisation with `release_all` – statement `f.get` –, the method `release_all` can be stopped at any point of its execution, thus making the set of released virtual machines non-determinate. In order to ban methods like `ugly_release`, we constrain definitions as follows:

1. every method invocation is synchronized within caller's body. In this way every effect of a method is computed before its termination.

2. it is not possible to release a machine that is executing a method.

Virtual machines' identity. Removals of virtual machines may have side effects on other machines. The following method `double_release` illustrates the point:

```
Int double_release(VM x, VM y) {
  release x; release y;
  return 0 ;
}
```

This method takes two machines and simply releases them. This means that, when the virtual machines in input are active *and different*, the cost of `double_release` is `-2`. In fact, this is the case of the following method `user1`:

```
Int user1() {
  VM x ; VM y ; Fut<Int> f ;
  x = new vm ; y = new vm;
  f = this!double_release(x, y);
  f.get ; return 0 ; }
```

```
Int user2() {
  VM x ; Fut<Int> f ;
  VM x = new vm ;
  f = this!double_release(x, x) ;
  f.get ; return 0 ; }
```

An easy computation returns a (net) cost of `0` for `user1`. However, the cost of `double_release` is *not always* `-2`. For example, consider the `user2` above. This method creates one virtual machine (cost `+1`) and invokes `double_release` with a duplicated argument: in this case the cost of `double_release` is `-1`, not `-2`, (and the cost of `user2` is `0`). Said more explicitly, the cost of a method depends on the identity of its arguments, not only on their states (alive or dead).

It is also worth to notice that the identity of arguments has impact on method's effects. Consider the following code snippet:

```
VM x = new vm; VM y = new vm;
x!double_release(x,y);
```

This code creates two new virtual machines and releases them by invoking `double_release`. The awkward point is that the callee machine coincides with the first argument. Therefore `double_release` will fail to release the second argument `y`.

In order to comply with these issue we annotate every operation that may have side effects with *identity sharing informations*, noted Θ in Section 4 – and generate a different cost function for every Θ. For instance, we have two (net) cost functions for the `double_release` method:

1. one where we consider `x` and `y` to be different: here its cost will be `-2` when the two machines are alive,

2. one where se consider the two parameters to be equal: here its cost will be `-1` when the two machines are alive.

In addition, in order to avoid the problematic issue of the code above, we forbid the callee machine to have the same identity of the (other) arguments of the invocation.

Release of carrier's machine. To simplify our analysis, we admit releases of the `this` machine – the carrier – to be the last statement (before `return`). While this restriction may be easily dropped, it avoids duplications of rules in Figure 3 to deal with the fact that the virtual machine `this` could be dead. In any case, we notice that this constraint does not affect the expressive power of the language.

4 The behavioral type system of `vml`

Our analysis uses abstract descriptions, called *behavioural types*, which are intermediate codes highlighting the features of `vml` programs that are relevant for the analysis of resources in Section 5. These types support compositional reasonings and are associated to programs by means of a type system.

The syntax of behavioural types uses *vm names* α, β, γ, $\cdots$, and *future names* f, f', $\cdots$. Sets of vm names will be ranged over by $\mathtt{S}$, $\mathtt{S}'$, $\mathtt{R}$, $\cdots$, and sets of future names will be ranged over by $\mathtt{F}$, $\mathtt{F}'$, $\cdots$. The syntactic rules are the following

$$
\begin{array}{rcll}
\mathbb{b} & ::= & \_ \;\mid\; \alpha & \text{basic value} \\
\mathbb{t} & ::= & \alpha \;\mid\; \partial \;\mid\; \bot \;\mid\; \top & \text{vm value} \\
se & ::= & \textit{integer constant} \;\mid\; x \;\mid\; (\mathbb{t} \leq \bot) \;\mid\; (\mathbb{t} \leq \top) \;\mid\; se\; \mathtt{op}'\; se & \text{size expression} \\
\mathtt{op}' & ::= & + \;\mid\; - \;\mid\; = \;\mid\; \leq \;\mid\; \geq \;\mid\; \wedge \;\mid\; \vee & \text{linear operation} \\
\mathbb{r}, \mathbb{s} & ::= & \mathbb{b} \;\mid\; se & \text{typing value} \\
\mathbb{z} & ::= & (\mathbb{b}, \alpha, \Theta, \mathtt{R}) \;\mid\; \mathbb{b} & \text{future value} \\
\mathbb{x} & ::= & \_ \;\mid\; \mathtt{F}\mathbb{t} \;\mid\; f \;\mid\; \mathbb{z} & \text{extended value} \\
\mathbb{a} & ::= & \mathtt{0} \;\mid\; \nu\alpha[\Theta] \;\mid\; \alpha^{\checkmark}[\Theta] \;\mid\; \nu f : \mathtt{m}\; \alpha(\overline{\mathbb{s}}) \;\mid\; f^{\checkmark}[\Theta, \mathtt{R}] & \text{atom} \\
\mathbb{c} & ::= & \mathbb{a} \rhd \Gamma \;\mid\; \mathbb{a} \,⨟\, \mathbb{c} \;\mid\; \mathbb{c} + \mathbb{c} \;\mid\; (se)\{\mathbb{c}\} & \text{behavioural type}
\end{array}
$$

Behavioural types express creations of virtual machines ($\nu\alpha$) and their removal ($\alpha^{\checkmark}$), method invocations ($\nu f : \mathtt{m}\; \alpha(\overline{\mathbb{s}})$) and corresponding retrieval of the value ($f^{\checkmark}[\Theta, \mathtt{R}]$) and the conditionals (respectively $(se)\{\mathbb{c}\} + (\neg se)\{\mathbb{c}'\}$ or $\mathbb{c} + \mathbb{c}'$, according to the boolean guard is a size expressions that depends on the arguments of a method or not). Behavioural types also carry *vm name environments* Θ, Θ', $\cdots$. These environments map vm names to extended values $\mathtt{F}\mathbb{t}$, which are called *vm states* in the following. This feature is new in behavioural types and, as discussed in Section 3, it is needed during the analysis to manage the identity of methods' arguments. We will provide additional examples in the rest of the paper to explain vm name environments and their use.

In order to have a more precise type of continuations, the leaves of behavioural types are labelled with *environments*, ranged over by Γ, Γ', $\cdots$. Environments are maps from method names $\mathtt{m}$ to terms $\alpha(\overline{\mathbb{r}}) : \mathbb{b}, \mathtt{R}$, from variables to extended values $\mathbb{x}$, from future names to future values, and from vm names to vm states. These environments are used in the typing proofs and are dropped in the final types (method types and the main statement type).

The type system uses judgments of the following form:

– $\Gamma \vdash e : \mathbb{x}$ for pure expressions e, $\Gamma \vdash f : \mathbb{z}$ for future names f, and $\Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{r}}) : \mathbb{m}, \mathtt{R}$ for methods.

– $\Gamma \vdash_{\mathtt{S}} z : \mathbb{x}, \mathbb{c} \triangleright \Gamma'$ for expressions with side effects z, where $\mathbb{x}$ is the value, $\mathbb{c}$ is the behavioural type for z and Γ' is the environment Γ *with updates* of variables and future names.

– $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c}$, in this case the updated environments are inside the behavioural type Γ', in correspondence of every branch of its.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. Moreover we define

$$\Gamma[x \mapsto \mathbb{x}](y) \overset{def}{=} \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases} \qquad \Gamma|_X(x) \overset{def}{=} \begin{cases} \Gamma(y) & \text{if } x \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

The following operation and notations are going to be used:

– vm values $\mathbb{t}$ are partially ordered by the relation $\leq$ defined by $\partial \leq \top$ and $\partial \leq \bot$. In the following we will use the partial operation $\mathbb{t} \sqcap \mathbb{t}'$ returning, whenever it exists, the greatest lower bound between $\mathbb{t}$ and $\mathbb{t}'$. For example $\top \sqcap \bot = \partial$, but $\partial \sqcap \alpha$ is not defined.

– the *update of a vm name environment* Θ with respect to f and Γ (we remind that Θ is defined on vm names only), written $\Theta \searrow^{f} \Gamma$, returns a vm name environment defined as follows:

$$\Theta \searrow^{f} \Gamma \overset{def}{=} [\alpha \;\mapsto\; (\mathtt{F}' \setminus \{f\})(\mathbb{t} \sqcap \mathbb{t}')]^{\alpha \in \mathrm{dom}(\Theta), \Theta(\alpha) = \mathtt{F}\mathbb{t}, \Gamma(\alpha) = \mathtt{F}'\mathbb{t}'}$$

– the *multihole contexts* $\mathcal{C}[\,]$ defined by the following syntax:

$$\mathcal{C}[\,] \;::=\; [\,] \;\mid\; \mathbb{a} \,⨟\, \mathcal{C}[\,] \;\mid\; \mathcal{C}[\,] + \mathcal{C}[\,] \;\mid\; (se)\{\mathcal{C}[\,]\}$$

and, whenever $\mathbb{c} = \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n]$, then $\mathbb{c}[x \mapsto \mathbb{x}]$ is defined as $\mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1[x \mapsto \mathbb{x}]] \cdots [\mathbb{a}_n \triangleright \Gamma_n[x \mapsto \mathbb{x}]]$.

The type systems for expressions, expressions with side effects and statements are reported in Figures 2 and 3. It is worth to notice that the type system for expressions is not standard because (size) expressions containing method's arguments are typed with the expressions themselves. This is crucial in the cost analysis of Section 5. As regards the rules for statements, we discuss (T-Invoke), (T-Get), (T-Release), and (T-New) because the other ones are straightforward.

Rule (T-Invoke) types method invocations $e!\mathtt{m}(\overline{e})$ by means of a new future name f that is associated to the method name, the vm name of the callee and the arguments. The relevant point is the value of f in the updated environment.

(T-Var)
$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

(T-Primitive)
$$\Gamma \vdash \mathtt{p} : \mathtt{p}$$

(T-Op)
$$\frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1 \ \mathtt{op}' \ e_2 : se_1 \ \mathtt{op}' \ se_2}$$

T-Unit
$$\frac{\Gamma \vdash e : se}{\Gamma \vdash e : \_}$$

(T-Op-Unit)
$$\frac{\Gamma \vdash e_1 : \_ \quad \mathtt{or} \quad \Gamma \vdash e_2 : \_ \quad \mathtt{or} \quad \mathtt{op} \in \{*, /\}}{\Gamma \vdash e_1 \ \mathtt{op} \ e_2 : \_}$$

(T-Pure)
$$\frac{\Gamma \vdash e : \mathbb{x}}{\Gamma \vdash e : \mathbb{x}, \mathtt{0} \triangleright \Gamma}$$

(T-Method-Sig)
$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{\mathbb{r}}) : \mathbb{O}, \mathtt{R} \quad \overline{\beta} \subseteq fv(\alpha, \overline{\mathbb{r}}, \mathbb{O}) \\ \sigma \text{ is a vm renaming such that } \mathbb{O} \notin fv(\alpha, \overline{\mathbb{r}}) \text{ implies } \sigma(\mathbb{O}) \text{ fresh}\end{array}}{\Gamma \vdash \mathtt{m} \ \sigma(\alpha)(\sigma(\overline{\mathbb{r}})) : \sigma(\mathbb{O}), \sigma(\mathtt{R})}$$

Figure 2: Typing rules for expressions

This value contains the returned value, the vm name of the callee, a vm name environment and the set of vm names that the method is going to remove. The vm name environment records the state of vm names when the method is invoked and it will be used when the method is synchronized to update the environment of the synchronisation (see rule (T-Get)) with the changes performed by methods in parallel. It is important to observe that the environment returned by the judgment is updated with informations about vm names released by the method: every such name will contain f in its state. Next we discuss the constraints in the second line and third line of the premise of (T-Invoke). Assuming that the callee has not been already released ($\Gamma(\alpha) \neq \mathtt{F}\bot$), there are two cases:

(i) either $\Gamma(\alpha) = \varnothing\top$ or α is the caller object, namely the callee is alive because it has been created by the caller or it is the caller itself,

(ii) or $\Gamma(\alpha) \neq \varnothing\top$. This case has two subclases, namely either ($ii.a$) the callee is being released by a parallel method or ($ii.b$) it is an argument of the caller method – see rule (T-Method).

While in (i) we admit that the invoked method releases vm names, in case (ii) we forbid any release because the nondeterminism of the execution makes the analysis too much imprecise (in our view). In facts, as discussed in Section 3, in case ($ii.a$), we cannot determine what subset of $\mathtt{R}$ will be actually released (because of the parallel method releasing α). In case ($ii.b$), being α an argument of the method, it may retain any state when the method is invoked and, for reasons similar to ($ii.a$), it is not possible to determine at static time the exact subset of $\mathtt{R}$ that will be released. It is worth to notice that the constraint $\mathtt{R} = \varnothing$ of case (ii) enforces a programming style that reduces uncertainty and provides a more precise cost analysis. The constraint in the third line of the premise of (T-Invoke) applies the same constraint of the callee to the other invocations in parallel and to the object executing $e!\mathtt{m}(\overline{e})$: it is not possible to remove a vm name that is a callee of a parallel method.

Rule (T-Get) defines the synchronisation with a method invocation that corresponds to a future f. Let $(\mathbb{O}, \alpha, \Theta, \mathtt{R})$ be the value of f in the environment.

(T-Invoke)
$$\frac{\begin{array}{c}\Gamma \vdash e : \alpha \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{s}} \qquad \Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{s}}) : \mathbb{o}, \mathtt{R} \qquad \Gamma \vdash \mathtt{this} : \alpha' \\ \Gamma(\alpha) \neq \mathtt{F}\bot \quad \text{and} \quad ((\Gamma(\alpha) \neq \varnothing\top \text{ and } \alpha \neq \alpha') \quad \text{implies} \quad \mathtt{R} = \varnothing) \\ \mathtt{R} \cap \Big(\{\alpha'\} \cup \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \Theta, \mathtt{R}')\}\Big) = \varnothing \\ f \text{ fresh} \qquad \Gamma' = \Gamma[\beta \mapsto (\{f\} \cup \mathtt{F}')\mathbb{t}]^{\beta \in \mathtt{R}, \Gamma(\beta) = \mathtt{F}'\mathbb{t}}\end{array}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \nu f \colon \mathtt{m}\ \alpha(\overline{\mathbb{s}}) \rhd \Gamma'[f \mapsto (\mathbb{o}, \alpha, \Gamma|_{\mathtt{S} \cup fv(\alpha, \overline{\mathbb{s}})}, \mathtt{R})]}$$

(T-Invoke-Bot)
$$\frac{\begin{array}{c}\Gamma \vdash e : \alpha \qquad \Gamma(\alpha) = \mathtt{F}\bot \\ f \text{ fresh}\end{array}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \mathtt{0} \rhd \Gamma'[f \mapsto \_]}$$

(T-New)
$$\frac{\beta \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} \mathtt{new\ vm} : \beta, \nu\beta[\Gamma|_{\mathtt{S}}] \rhd \Gamma[\beta \mapsto \varnothing\top]}$$

(T-Get)
$$\frac{\begin{array}{c}\Gamma \vdash x : f \qquad \Gamma \vdash f : (\mathbb{o}, \alpha, \Theta, \mathtt{R}) \\ \Theta' = \Theta \searrow^{f} \Gamma \qquad \mathtt{R}' = fv(\mathbb{o}) \setminus (\mathtt{R} \cup \mathrm{dom}(\Theta)) \\ \Gamma' = \Gamma[\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \Theta'(\alpha)]^{\beta' \in \mathtt{R}'}[f \mapsto \mathbb{o}]\end{array}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, f^{\checkmark}[\Theta', \mathtt{R}] \rhd \Gamma'}$$

(T-Get-Done)
$$\frac{\Gamma \vdash x : f \qquad \Gamma \vdash f : \mathbb{o}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, \mathtt{0} \rhd \Gamma}$$

(T-Release)
$$\frac{\Gamma \vdash x : \alpha}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}(x) : \alpha^{\checkmark}[\Gamma|_{\mathtt{S} \cup \{\alpha\}}] \rhd \Gamma[\alpha \mapsto \varnothing\bot]}$$

(T-Assign-Var)
$$\frac{\Gamma(x) = \mathbb{x} \qquad \Gamma \vdash_{\mathtt{S}} z : \mathbb{x}', \mathbb{c}}{\Gamma \vdash_{\mathtt{S}} x = z : \mathbb{c}[x \mapsto \mathbb{x}']}$$

(T-If)
$$\frac{\Gamma \vdash e : se \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \} : (se)\{\ \mathbb{c}_1\ \} + (\neg se)\{\ \mathbb{c}_2\ \}}$$

(T-If-ND)
$$\frac{\Gamma \vdash e : \_ \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \} : \mathbb{c}_1 + \mathbb{c}_2}$$

(T-Seq)
$$\frac{\Gamma \vdash_{\mathtt{S}} s_1 : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \qquad \Gamma_i \vdash_{\mathtt{S}} s_2 : \mathbb{c}_i'}{\Gamma \vdash_{\mathtt{S}} s_1; s_2 : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{c}_1'] \cdots [\mathbb{a}_n \fatsemi \mathbb{c}_n']}$$

(T-Return)
$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbb{o} \qquad \Gamma \vdash \mathtt{destiny} : \mathbb{o}' \\ \mathbb{o} \in \mathtt{S} \cup \{\_\} \quad \text{if and only if} \quad \mathbb{o} = \mathbb{o}'\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{return}\ e : \mathtt{0} \rhd \Gamma[\mathtt{destiny} \mapsto \mathbb{o}]}$$

Figure 3: Type rules for expressions with side effects and statements.

Since R defines the resources of the caller that are released, we record in the returned environment Γ' that these resources are no more available. Γ' also records the state of the returned vm name, if it is a virtual machine that has been created by the method of f. This state is the same of the callee vm name (which may have been updated since the invocation), namely the value of $(\Theta \searrow^{f} \Gamma)(\alpha)$. As regards the behavioural type $f^{\checkmark}[\Theta \searrow^{f} \Gamma, \mathtt{R}]$ of $x.\mathtt{get}$, it carries two arguments (i) the vm name environment of the invocation updated with the invocation of parallel methods and (ii) the vm names to be released. These two arguments will let us disable the removal of names in the cost analysis when these names are removed by methods in parallel. In this case, the removal is computed in the caller method (and therefore it counts `-1`, instead of `-1` in every method in parallel) – see definition of `translate` in case of $f^{\checkmark}[\Theta \searrow^{f} \Gamma, \mathtt{R}]$ in Section 5.

Rule (T-release) models the removal of a vm name α. Notice that the behavioural type is not just $\alpha^{\checkmark}$, which should correspond to a `-1` in the cost analysis. Rather, it is a “conditional” removal because the name may be removed by some method in parallel, or even may have been already removed when the method has been invoked. This is the reason for the presence of the vm name environment $[\Gamma|_{\mathtt{S}\cup\{\alpha\}}]$. We also observe that this rule applies only when the caller has not been already released.

For reasons similar to the rules discussed above, (T-New) adds an environment $\Gamma|_{\mathtt{S}}$ to the type $\nu\beta$ expressing a new resource. In facts, this environment will make the cost increase by `1` provided the caller method has not been already released – see definition of `translate` in case of $\nu\beta[\Gamma|_{\mathtt{S}}]$ in Section 5.

The type system of `vml` is completed with the rules for method declarations and programs:

(T-Method)

$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \mathbb{o}, \mathtt{R} \qquad \mathtt{S} = \{\alpha\} \cup \overline{\beta} \\ \Gamma[\mathtt{this} \mapsto \alpha][\mathtt{destiny} \mapsto \mathbb{o}][\overline{x} \mapsto \overline{x}][\overline{z} \mapsto \overline{\beta}][\alpha \mapsto \varnothing\alpha][\overline{\beta} \mapsto \varnothing\overline{\beta}] \vdash_{\mathtt{S}} s : \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n] \\ \Big(\Gamma_i(\gamma) = \Gamma_j(\gamma)\Big)^{i,j\in 1..n,\ \gamma\in\mathtt{S}\cup fv(\mathbb{o})} \qquad \mathtt{R} = (\mathtt{S} \cup fv(\mathbb{o})) \cap \{\gamma \mid \Gamma_1(\gamma) = \bot\}\end{array}}{\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{Int}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\}\ : \mathtt{m}\ \alpha(\overline{x},\overline{\beta})\ \{\, \mathcal{C}[\mathbb{a}_1 \triangleright \varnothing] \cdots [\mathbb{a}_n \triangleright \varnothing]\,\} : \mathbb{o},\ \mathtt{R}}$$

(T-Program)

$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \qquad \Gamma \vdash_{\mathsf{start}} s : \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\overline{F\ x\ ;}\ s\} : \overline{\mathbb{C}},\ \mathcal{C}[\mathbb{a}_1 \triangleright \varnothing] \cdots [\mathbb{a}_n \triangleright \varnothing]}$$

Without loss of generality, rule (T-Method) assumes that formal parameters of methods are ordered: those of `Int` type occur before those of `Vm` type. We observe that the environment typing the method body binds integer parameters to their same name, while the other ones are bound to fresh vm names (this lets us to have a more precise cost analysis in Section 5). We also observe that the returned value $\mathbb{o}$ may be either $\_$ or a vm name in `S` or a fresh vm name. In this last case, the premise $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j\in 1..n,\ \gamma\in\mathtt{S}\cup fv(\mathbb{o})}$ guarantees that every branch in the returned behavioural type creates a new vm name and, by rule (T-Return), the chosen vm name must be always the same.

We display behavioural types at work by typing codes of Section 2 and 3. Actually, the following types do not abstract a lot from codes' details because the programs of the previous sections have been designed for highlighting the issues of our technique.

The behavioural types of `fact` and `cheap_fact` are the following ones

```
fact $\alpha$(n) {
  (n==0){ 0 }
  + (n>0){ $\nu\beta[\alpha \mapsto \varnothing\alpha]$ ⨾
              $\nu\, x : \mathtt{fact}\, \beta(n-1)$ ⨾
              $x^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top]$ ⨾
              $\beta^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top]$ ⨾ }
} - , { }
```

```
cheap_fact $\alpha$(n, -) {
  (n==0){ 0 }
  + (n>0){ $\nu\beta[\alpha \mapsto \varnothing\alpha]$ ⨾
              $\nu\, x : \mathtt{prod}\, \beta(n,$ -) ⨾
              $x^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top]$ ⨾
              $\beta^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top]$ ⨾
              $\nu\, y : \mathtt{cheap\_fact}\, \beta(n-1,$ -) ⨾
              $y^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\bot]$ ⨾ }
} - , { }
```

We notice that the types record the order between recursive invocations and releases of machines, which is different in the two codes for factorial.

The behavioural types of `double_release` and `user1` are the following ones

`double_release` $\alpha(\beta,\ \gamma)$ `{`
  $\beta^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta, \gamma \mapsto \varnothing\gamma] \fatsemi$
  $\gamma^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\bot, \gamma \mapsto \varnothing\gamma] \fatsemi$
`} - ,` $\{\beta,\ \gamma\}$

`user1` $\alpha($ `){`
  $\nu\beta[\alpha \mapsto \varnothing\alpha]$; $\nu\gamma[\alpha \mapsto \varnothing\alpha] \fatsemi$
  $\nu\, f : \mathtt{double\_free}\, \alpha(\beta,\gamma) \fatsemi$
  $f^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top, \gamma \mapsto \varnothing\top] \fatsemi$
`} - , { }`

It is worth to notice that the releases $\beta^{\checkmark}$ and $\gamma^{\checkmark}$ in `double_release` are conditioned by the values of β and γ when the method is invoked. In facts, in case of `user1` these values are $\varnothing\top$ and this will mean that the cost of $f^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top, \gamma \mapsto \varnothing\top]$ will be `-2`.

5 The analysis of behavioural types

The behavioural types returned by system in Section 4 are used to compute the resource elasticity of a `vml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [10] – and, in this section, we discuss the translation of a behavioural type program into a set of *recurrence relations* that fed the solver.

Basically, our translation maps method types into functions from parameters to cost, where

- method invocations are translated into function calls,
- virtual machine creations are translated into a `+1` cost,
- virtual machine releases are translated into a `-1` cost,

There are two function calls for every method invocation: one returns the maximal number of resources needed to execute a method `m`, called *peak cost* of `m` and noted $\mathtt{m}_{\text{peak}}$, and the other returns the number of resources the method `m` creates without releasing, called *net cost* of `m` and noted $\mathtt{m}_{\text{net}}$. These functions are used to define the cost of sequential execution and parallel execution of methods. For example, omitting arguments of methods, the cost of the sequential composition of two methods `m` and $\mathtt{m}'$ is the maximal value between $\mathtt{m}_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{net}}$; while the cost of the parallel execution of `m` and $\mathtt{m}'$ is the maximal value between $\mathtt{m}_{\text{peak}} + \mathtt{m}'_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{net}}$.

There are two difficulties that entangle our translation that pertain to method invocations: the management of arguments' identities and of argument's values.

Argument's identities. Consider the code

```
Int free() { release(this) ; return(0) ; }

Int m(VM x, VM y) {
  Fut<Int> f ; f = x!free() ;
  release y ; f.get ; return(0) ;
}
```

The behavioural types of these methods are

$$\begin{array}{l}\texttt{free}\ \alpha(\ )\{\ \alpha^{\checkmark}[\alpha \mapsto \varnothing\top]\ \}\ \_, \{\alpha\} \\ \texttt{m}\ \alpha(\beta,\gamma)\{\nu\, f : \texttt{free}\,\beta(\ )\ ⨾\ \gamma^{\checkmark}[\Theta]\ ⨾\ f^{\checkmark}[\Theta']\}\ \_, \{\beta,\gamma\}\end{array}$$

where

$$\Theta = \alpha \mapsto \varnothing\alpha, \beta \mapsto \{\texttt{free}\}\beta, \gamma \mapsto \varnothing\gamma$$
$$\Theta' = \alpha \mapsto \varnothing\alpha, \beta \mapsto \{\texttt{free}\}\beta, \gamma \mapsto \varnothing\bot$$

We notice that, in the type of `m`, there is not enough information to determine whether $\gamma^{\checkmark}$ will have a cost equal to `-1` or `0`. In facts, while in typing rules of methods the arguments are assumed to be pairwise different – see rule (T-Method) –, it is not the case for invocations. For instance, if `m` is invoked with two arguments that are equal – $\beta = \gamma$ – then γ is going to be released by the invocation $\texttt{free}(\beta)$ and therefore it counts `0`. To solve this problem of arguments' identity, we refine even more the translation of method, which now depends on an equivalence relation telling which of the vm names in parameter are actually equal or not. Hence, in general, our method `m` will be translated in four cost functions: $\texttt{m}_{\text{peak}}^{\{1\},\{2\}}(x,y)$ and $\texttt{m}_{\text{net}}^{\{1\},\{2\}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\texttt{m}_{\text{peak}}^{\{1,2\}}(x)$ and $\texttt{m}_{\text{net}}^{\{1,2\}}(x)$, which correspond to the invocations where $x = y$. (The equivalence relation in the superscript never mention `this`, which is also an argument, because we assume that `this` is always different from the other arguments, see below.)

The following function `EqRel` computes the equivalence relation corresponding to a specific method call; `EqRel` takes a tuple of vm names and returns an equivalence relation on indices of the tuple:

$$\texttt{EqRel}(\alpha_1,\cdots,\alpha_n) \ = \ \bigcup_{i\in 1..n} \{\ \{j \mid \alpha_j = \alpha_i\}\ \}$$

Let $\texttt{EqRel}(\alpha_1,\cdots,\alpha_n)(\beta_1,\cdots,\beta_n)$ be the tuple $(\beta_{i_1},\cdots,\beta_{i_k})$, where $i_1,\cdots,i_k$ are *canonical representatives* of the sets in $\texttt{EqRel}(\alpha_1,\cdots,\alpha_n)$ (we take the vm name with the least index in every set). We observe that, by definition, $\texttt{EqRel}(\alpha_1,\cdots,\alpha_n)(\alpha_1,\cdots,\alpha_n)$ is a tuple of pairwise different vm names (in $\alpha_1,\cdots,\alpha_n$).

We will always assume that $\{1\} \in \texttt{EqRel}(\alpha_1,\cdots,\alpha_n)$. Since α_1 always represents the `this` object, this constraint enforces that the other arguments of invocations are always different from `this` (see the discussion in the paragraph about virtual machines identity in Section 3).

Argument's values. Consider the code

```
Int m(VM x) {
  Fut<Int> f ; Fut<Int> g ; VM y ; VM z ; Int u ;
  y = new vm ; z = new vm ;
  f = x!free() ; g = y!foo(x, z) ;
  u = g.get ; u = f.get ; return 0 ;
}
```

where `free` is as above and `foo` is unspecified (we assume that `foo` does not release any machine). The behavioural types of `m` is

$$\texttt{m}\,\alpha(\beta)\{\ \nu\gamma[\Theta_\gamma] \fatsemi \nu\gamma'[\Theta_{\gamma'}]; \nu\, f : \texttt{free}\,\beta(\ ) \fatsemi \nu\, g : \texttt{foo}\,\gamma(\beta,\gamma') \fatsemi g^{\checkmark}[\Theta_2]; f^{\checkmark}[\Theta_1]\}$$
$$\_\,,\{\beta\}$$

where
$$\begin{array}{rcl}
\Theta_\gamma & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta] \\
\Theta_{\gamma'} & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta, \gamma \mapsto \varnothing\top] \\
\Theta_2 & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \{f\}\beta, \gamma \mapsto \varnothing\top, \gamma' \mapsto \varnothing\top] \\
\Theta_1 & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \{f\}\beta, \gamma \mapsto \varnothing\top, \gamma' \mapsto \varnothing\top]
\end{array}$$

The arguments of the invocation of `foo` are β and γ' and, in order to compute its cost, it is necessary to instantiate these arguments with actual vm values. This instantiation is straightforward for γ': there is no concurrent future releasing it, so its state is $\top$. However, this is not the case for β': as specified in Θ_2, its vm state is tagged with a nonempty set of futures because `free` is concurrently releasing it. Hence, we need to translate this state into a simple one ($\bot$, $\top$, or ∂). In facts, the value of the first argument of `foo` may be ∂ if β was originally either $\top$ or ∂, or it may be $\bot$ if β was already released. To deal with this case, we introduce an operator on vm values $\beta\downarrow$ whose meaning is the one just described:

$$\beta\downarrow = \begin{cases} \partial & \text{if } \beta = \top \text{ or } \beta = \partial \\ \bot & \text{else} \end{cases}$$

Since it is not possible to know the value of β during the translation, we use $\beta\downarrow$ in the syntax of our cost expressions. This term will be evaluated during the solving process.

To conclude, the translation of vm states into vm values (extended with $\alpha\downarrow$), written $\texttt{F}\mathbb{t}\Downarrow$, is defined as follows:

$$\texttt{F}\mathbb{t}\Downarrow \stackrel{def}{=} \begin{cases} \mathbb{t} & \text{if } \texttt{F} = \varnothing \\ \partial & \text{if } \texttt{F} \neq \varnothing \text{ and } \mathbb{t} = \top \\ \bot & \text{if } \texttt{F} \neq \varnothing \text{ and } \mathbb{t} = \bot \\ \alpha\downarrow & \text{if } \texttt{F} \neq \varnothing \text{ and } \mathbb{t} = \alpha \end{cases}$$

and we write $(\texttt{F}_1\mathbb{t}_1, \cdots, \texttt{F}_n\mathbb{t}_n)\Downarrow$ for $(\texttt{F}_1\mathbb{t}_1\Downarrow, \cdots, \texttt{F}_n\mathbb{t}_n\Downarrow)$.

The translation function. The translation function, called `translate`, is structured in three parts that respectively correspond to simple atoms, full behavioral types, and method types and full programs. `translate` carries five arguments:

1. Δ is the *equivalence relation on formal parameters* identifying those that are equal. We assume that $\Delta(x)$ returns the unique representative of the equivalence class of x. Therefore we can use Δ also as a substitution operation.

2. Ψ is the *translation environment* which stores temporary information about futures that are active (unsynchronized);

3. α is the vm name of the virtual machine of the current behaviour type;

4. $(se)\{\,\overline{e}\,\}$ is sequence of costs of the current execution branch: the size expression se stores all the conditions corresponding to the current execution branch, while $\overline{e}$ is the sequence of (over-approximated) costs that branch takes during its execution.

5. the behavioural type that is translated; it may be either $\mathbb{a}$, $\mathbb{c}$ or $\overline{\mathbb{C}}$.

The auxiliary functions below let us reduce the number of cases of the definition of `translate`:

$$\texttt{CNEW}(\alpha) \;=\; \begin{cases} \texttt{0} & \alpha = \bot \\ \texttt{1} & \text{otherwise} \end{cases}$$

This function is used when a virtual machine is created. It returns `1` or `0` according to the virtual machine that is executing the code can be alive ($\alpha \neq \bot$) or not, respectively.

$$\texttt{CREL}(\alpha) \;=\; \begin{cases} \texttt{-1} & \alpha = \top \\ \texttt{0} & \text{otherwise} \end{cases}$$

This function is used when a virtual machine is released (in correspondence of atoms $\beta^{\checkmark}$). The release is effectively computed – value `-1` – only when the virtual machine that is executing the code is alive ($\alpha = \top$).

The `translate` function also uses the *merge operation*, noted $\Theta[\Delta]$, that takes a vm name environment Θ and an equivalence relation on vm names Δ and returns a *substitution*. We remind that vm name environments have been introduced in Section 4 to manage identities of arguments in method calls. Take, for instance, the atom $f^{\checkmark}[\Theta, \mathtt{R}]$ within a behavioural type that binds f to $\mathtt{foo}\,\alpha(\beta,\gamma)$. Assume to evaluate this behavioural type for $\mathtt{m}^{\Delta}_{\text{peak}}$ where $\Delta = \{\beta,\gamma\}$. That is, the two arguments are actually identical. What are the values of β and γ for evaluating $\mathtt{foo}^{\Delta}_{\text{peak}}$ and $\mathtt{foo}^{\Delta}_{\text{net}}$? Well, we have

1. to select the representative between β and γ: it will be $\Delta(\beta)$ (which is equal to $\Delta(\gamma)$;

2. to take a value that is smaller than $\Theta(\beta)$ and $\Theta(\gamma)$ (but greater than any other value that is smaller);

3. to substitute β and γ with the result of 2.

For example, let $\Theta = [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta, \gamma \mapsto \varnothing\gamma]$ and $\Delta(\beta) = \Delta(\gamma) = \beta$. We expect that a value for the item 2 above is $\varnothing\beta$ and the substitution of the item 3 is $\{^{\varnothing\beta,\varnothing\beta}/_{\beta,\gamma}\}$. Formally, the operation returning the value for 2 is noted $\otimes^{\beta}$

below (it applies to vm values and vm states) and the operation returning the substitution of item 3 is the merge operation.

$$\mathbb{t} \otimes^{\alpha} \mathbb{t}' \;=\; \begin{cases} \bot & \text{if } \mathbb{t} = \bot \text{ or } \mathbb{t}' = \bot \\ \alpha\downarrow & \text{if } (\mathbb{t} = \gamma\downarrow \text{ and } \mathbb{t}' \neq \bot) \text{ or } (\mathbb{t} \neq \bot \text{ and } \mathbb{t}' = \gamma\downarrow) \\ \alpha & \text{if } \mathbb{t} = \gamma \text{ and } \mathbb{t}' = \gamma' \end{cases}$$

$$\mathtt{F}_1\mathbb{t}_1 \otimes^{\alpha} \mathtt{F}_2\mathbb{t}_2 \;=\; (\mathtt{F}_1 \cup \mathtt{F}_2)(\mathbb{t}_1 \otimes^{\alpha} \mathbb{t}_2)$$

$$\Theta[\Delta] \;:\; \Big[\alpha \mapsto \bigotimes^{\Delta(\alpha)}\{\Theta(\beta) \mid \beta \in \mathrm{dom}(\Theta) \text{ and } \Delta(\beta) = \Delta(\alpha)\}\Big]^{\alpha\in\mathrm{dom}(\Theta)}$$

We notice that the definition of $\otimes^{\alpha}$ is not necessary for vm values as ∂ or $\top$ because we merge vm names whose image by Θ can only be either $\mathtt{F}\beta$ or $\mathtt{F}\beta\downarrow$ or $\mathtt{F}\bot$. As a notational remark, we observe that the substitution $\Theta[\Delta]$ is noted as a map $[\alpha_1 \cdots \mathtt{F}_1\mathbb{t}_1, \cdots, \alpha_n \cdots \mathtt{F}_n\mathbb{t}_n]$ instead of the standard notation $\{{}^{\mathtt{F}_1\mathbb{t}_1,\cdots,\mathtt{F}_n\mathbb{t}_n}/_{\alpha_1,\cdots,\alpha_n}\}$. These two notations are clearly equivalent: we prefer the former one because it will let us to write $\Theta[\Delta](\alpha)$ or even $\Theta[\Delta](\alpha_1, \cdots, \alpha_n)$ with the obvious meanings.

Every preliminary notion is in place for defining `translate`. We begin with the translation of atoms.

$$\mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}; e\,\}, \mathbb{a}) \;=$$
$$\begin{cases}
(\Psi, (se)\{\,\overline{e}; e\,\}) & \text{when } \mathbb{a} = \mathtt{0} \\
(\Psi, (se)\{\,\overline{e}; e; e + \mathtt{CNEW}(\mathbb{t})\,\}) & \text{when } \mathbb{a} = \nu\beta[\Theta] \\
 & \text{and } \Theta[\Delta](\alpha) = \mathtt{F}\mathbb{t} \\
(\Psi, (se)\{\,\overline{e}; e; e + \mathtt{CREL}(\mathbb{t})\,\}) & \text{when } \mathbb{a} = \beta^{\checkmark}[\Theta] \\
 & \text{and } \Theta[\Delta](\beta) = \mathtt{F}\mathbb{t} \\
(\Psi[f \mapsto \mathtt{m}\ \Delta(\beta)(\overline{se}, \Delta(\overline{\beta}))], (se)\{\,\overline{e}; e; e + f\,\}) & \text{when } \mathbb{a} = (\nu f = \mathtt{m}\ \beta(\overline{se}, \overline{\beta})) \\
(\Psi \setminus f, (se)\{\,(\overline{e}; e)\sigma; (\overline{e}; e)\sigma' + \sum_{\gamma\in\Delta(\mathtt{R}),\Theta[\Delta](\gamma)=\mathtt{F}\mathbb{t},\mathtt{F}\neq\varnothing} \mathtt{CREL}(\mathbb{t})\,\} & \text{when } \mathbb{a} = f^{\checkmark}[\Theta, \mathtt{R}] \\
 & \text{where} \\
 & \Psi(f) = \mathtt{m}\ \beta(\overline{se}, \overline{\beta}) \\
 & \mathtt{EqRel}(\beta, \overline{\beta}) = \Xi \\
 & \sigma = \{{}^{\mathtt{m}^{\Xi}_{\mathrm{peak}}(\overline{se}, \Theta[\Delta](\Xi(\beta,\overline{\beta}))\Downarrow)}/_{f}\} \\
 & \sigma' = \{{}^{\mathtt{m}^{\Xi}_{\mathrm{net}}(\overline{se}, \Theta[\Delta](\Xi(\beta,\overline{\beta}))\Downarrow)}/_{f}\}
\end{cases}$$

In the definition of `translate` we always highlight the last expression in the sequence of costs of the current execution branch (the fourth input). This is because the cost of the parsed atom applies to it, except for the case of $f^{\checkmark}[\Theta, \mathtt{R}]$. In this last case, let $(se)\{\,\overline{e}; e\,\}$ be the expression. Since the atom expresses the synchronisation of f, $\overline{e}; e$ will have occurrences of f. In this case, the function `translate` has to compute two values: the maximum number of resources used by (the method corresponding to) f during its execution – the *peak cost* used in the substitution σ – and the resources used upon the termination of (the method corresponding to) f – the *net cost* used in the substitution σ'. In particular, this last value has to be decreased by the number of the resources released by the method. This the purpose of the addend $\sum_{\gamma\in\Delta(\mathtt{R}),\Theta[\Delta](\gamma)=\mathtt{F}\mathbb{t},\mathtt{F}\neq\varnothing}\mathtt{CREL}(\mathbb{t})$ that remove machines that are going to be removed by parallel methods (the constraint $\mathtt{F}\neq\varnothing$) because the other ones have been already counted both in the peak cost and in the net cost. We observe that the instances of the method $\mathtt{m}_{\rm peak}$ and $\mathtt{m}_{\rm net}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta,\overline{\beta})$.

The definition of `translate` for behavioural type is given below. It follows straightforwardly by composing the definitions of the atoms. We also observe that, in this case, the sequences of costs are sets.

$$
\begin{array}{l}
\mathtt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}) \;=\\
\qquad\begin{cases}
\{(se')\{\,\overline{e'}\,\}\} & \text{when } \mathbb{c}=\mathbb{a}\triangleright\varnothing \text{ and } \mathtt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',(se')\{\,\overline{e'}\,\})\\[2mm]
C'' & \text{when } \mathbb{c}=\mathbb{a}\fatsemi\mathbb{c}' \text{ and } \mathtt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',\{(se')\{\,\overline{e'}\,\}\})\\
 & \text{and } \mathtt{translate}(\Delta,\Psi',\alpha,(se')\{\,\overline{e'}\,\},\mathbb{c}')=(\Psi'',C'')\\[2mm]
C'\cup C'' & \text{when } \mathbb{c}=\mathbb{c}_1+\mathbb{c}_2 \text{ and } \mathtt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_1)=(\Psi',C')\\
 & \text{and } \mathtt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_2)=(\Psi'',C'')\\[2mm]
C' & \text{when } \mathbb{c}=(se')\{\mathbb{c}'\} \text{ and } \mathtt{translate}(\Delta,\Psi,\alpha,(se\wedge se')\{\,\overline{e}\,\},\mathbb{c}')=(\Psi',C')
\end{cases}
\end{array}
$$

The translation of method types and behavioural type programs is given below. Let $\mathcal{P}$ be the set of partitions of $1..n$. Then

$$
\mathtt{translate}(\mathtt{m}\;\alpha_1(\overline{x},\alpha_2,\dots,\alpha_n)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\mathtt{R})=\bigcup_{\Xi\in\mathcal{P}}\mathtt{translate}(\Xi,\mathtt{m}\;\alpha_1(\overline{x},\alpha_2,\dots,\alpha_n)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\mathtt{R})
$$

where $\mathtt{translate}(\Xi,\mathtt{m}\;\alpha_1(\overline{x},\alpha_2,\dots,\alpha_n)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\mathtt{R})$ is defined as follows. Let

$$
\Delta=\{\,\{\,\alpha_{i_1},\dots,\alpha_{i_m}\,\}\mid\{\,i_1,\dots,i_m\,\}\in\Xi\,\}
$$

and

$$
\mathtt{translate}(\Delta,\varnothing,\alpha_1,(\mathtt{true})\{\,0\,\},\mathbb{c})=\bigcup_{i=1}^{n}(se_i)\{\,e_{1,i};\dots;e_{h_i,i}\,\}
$$

Then

$$
\begin{array}{l}
\mathtt{translate}(\Xi,\mathtt{m}\;\alpha_1(\overline{x},\alpha_2,\dots,\alpha_k)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\mathtt{R}) \;=\\
\qquad\qquad\left\{\begin{array}{ll}
\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=0 & [\alpha_1=\bot]\\
\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{1,1} & [se_1\wedge\alpha_1\neq\bot]\\
\quad\vdots & \quad\vdots\\
\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{h_1,1} & [se_1\wedge\alpha_1\neq\bot]\\
\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{1,2} & [se_2\wedge\alpha_1\neq\bot]\\
\quad\vdots & \quad\vdots\\
\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{h_n,n} & [se_n\wedge\alpha_1\neq\bot]\\
\mathtt{m}^{\Xi}_{\rm net}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=0 & [\alpha_1=\bot]\\
\mathtt{m}^{\Xi}_{\rm net}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=\mathtt{m}^{\Xi}_{\rm peak}(\overline{x},\Xi[\alpha_1,\dots,\alpha_n]) & [\alpha_1=\partial]\\
\mathtt{m}^{\Xi}_{\rm net}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{n_1,1} & [se_1\wedge\alpha_1=\top]\\
\quad\vdots & \quad\vdots\\
\mathtt{m}^{\Xi}_{\rm net}(\overline{x},\Xi[\alpha_1,\alpha_2,\dots,\alpha_k])=e_{n_n,n} & [se_n\wedge\alpha_1=\top]
\end{array}\right.
\end{array}
$$

Let $(\mathbb{C}_1 \ \ldots \ \mathbb{C}_n, \ \mathbb{c})$ be a behavioural type program and let $\texttt{translate}(\varnothing, \varnothing, \alpha, (\texttt{true})\{\,0\,\}, \mathbb{c}) = \bigcup_{j=1}^{m} (se_j)\{\, e_{1,j}; \cdots ; e_{h_j,j} \,\}$. Then

$$\texttt{translate}(\mathbb{C}_1 \ \ldots \ \mathbb{C}_n, \ \mathbb{c}) \ = \ \begin{cases} \texttt{translate}(\mathbb{C}_1) \ \cdots \ \texttt{translate}(\mathbb{C}_n) & \\ \texttt{main}() = e_{1,1} & [se_1] \\ \quad\vdots & \vdots \\ \texttt{main}() = e_{h_1,1} & [se_1] \\ \texttt{main}() = e_{1,2} & [se_2] \\ \quad\vdots & \vdots \\ \texttt{main}() = e_{h_m,m} & [se_m] \end{cases}$$

We show the output of `translate` when applied to the behavioural types of `double_release`, `user1`, and `user2` that we have described in Section 4 (well, the behavioural type of `user2` has not been shown: it is left as an exercise). Since `double_release` has two arguments, we generate two sets of equations, as discussed above. On the contrary, methods `user1` and `user2` carry one argument and therefore there is one set of equations only. In order to ease the reading, we omit the equivalence classes of arguments that label function names: the reader may grasp them from the number of arguments. For the same reason, we represent a partition $\{\{1\},\{2\},\{3\}\}$ corresponding to vm names α_1, α_2 and α_3 by $[\alpha_1, \alpha_2, \alpha_3]$ and $\{\{1\},\{2,3\}\}$ by $[\alpha_1, \alpha_2]$ (we write the canonical representatives). For simplicity we do not add the partition to the name of the method.

$$\texttt{translate}([\alpha_1,\alpha_2,\alpha_3], \textit{double\_release}\ \alpha_1(\alpha_2,\alpha_3)\ \{\mathbb{c}_{\textit{double\_release}}\,\} : \_, \ \{\alpha_2,\alpha_3\}) \ =$$
$$\begin{cases} \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 \neq \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 \neq \bot] \\ & \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) & [\alpha_1 = \partial] \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 = \top] \end{cases}$$

$$\texttt{translate}([\alpha_1,\alpha_2], \textit{double\_release}\ \alpha_1(\alpha_2)\ \{\mathbb{c}_{\textit{double\_release}}\,\} : \_, \ \{\alpha_2\}) \ =$$
$$\begin{cases} \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 \neq \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CNEW}(\alpha_2) & [\alpha_1 \neq \bot] \\ \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 \neq \bot] \\ & \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = \textit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) & [\alpha_1 = \partial] \\ \textit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 = \top] \end{cases}$$

$$\texttt{translate}([\alpha_1], \mathit{user1}\ \alpha_1()\ \{\mathbb{c}_{user1}\} : \_,\ \{\}) =$$
$$\begin{cases}
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{peak}}(\alpha_1, \top, \top) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{net}}(\alpha_1, \top, \top) & [\alpha_1 \neq \bot] \\
\\
\mathit{user1}_{\mathrm{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\mathrm{net}}(\alpha_1) = \mathit{user1}_{\mathrm{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user1}_{\mathrm{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{net}}(\alpha_1, \top, \top) & [\alpha_1 = \top]
\end{cases}$$

$$\texttt{translate}([\alpha_1], \mathit{user2}\ \alpha_1()\ \{\mathbb{c}_{user2}\} : \_,\ \{\}) =$$
$$\begin{cases}
\mathit{user2}_{\mathrm{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\mathrm{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{peak}}(\alpha_1, \top) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\mathrm{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{net}}(\alpha_1, \top) & [\alpha_1 \neq \bot] \\
\\
\mathit{user2}_{\mathrm{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\mathrm{net}}(\alpha_1) = \mathit{user2}_{\mathrm{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user2}_{\mathrm{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\mathrm{net}}(\alpha_1, \top) & [\alpha_1 = \top]
\end{cases}$$

6 Outline of the proof of correctness

The proof of correctness of our technique is long even if almost standard (see [11] for a similar proof). In this section we overview it by highlighting the main difficulties.

The first part of the proof addresses the correctness of the type system in Section 4. As usual with type systems, the correctness is represented by a subject reduction theorem expressing that if a configuration *cn* of the operational semantics is well typed and $cn \to cn'$ then cn' is well-typed as well. It is worth to observe that we cannot hope to demonstrate a statement guaranteeing type-preservation (the types of *cn* and cn' are equal) because our types are "behavioural". However, it is critical for the correctness of the cost analysis that there is a relation between the type of *cn*, let it be $\mathbb{c}$, and the type of cn', let it be $\mathbb{c}'$.

Therefore, a subject reduction for the type system of Section 4 requires

1. the extension of the typing to configurations;
2. the definition of an evaluation relation $\rightsquigarrow$ between behavioural types.

Once 1 and 2 above have been defined, it is possible to demonstrate (let $\rightsquigarrow^*$ be the reflexive and transitive closure of $\rightsquigarrow$):

Theorem 6.1 (Subject Reduction). *Let cn be a configuration of a* `vml` *program and let* $\mathbb{c}$ *be its behavioural type. If* $cn \to cn'$ *then there is* $\mathbb{c}'$ *typing* cn' *such that* $\mathbb{c} \rightsquigarrow^* \mathbb{c}'$.

The proof of this theorem is by case on the reduction rule applied and it is usually not complex because the relation $\rightsquigarrow$ mimics the `vml` transitions in Section 2.

The second part of the proofs relies on the definition of the notion of *direct cost of a behavioural type* (of a configuration), which is the number of virtual machines occurring in the type. We also observe that the number of alive virtual machines in a configuration is identical to the direct cost of the corresponding a behavioural type. Then it is also necessary to define

3. the extension of the function `translate` to compute the recurrence relations for behavioural types of configurations. These recurrence relations let us compute the *peak cost of a behavioural type* (of a configuration).

The proofs of the following two properties are preliminary to the correctness of our technique:

Lemma 6.2 (Basic Cost Inclusion)**.** *The direct cost of a behavioural type of a configuration is less or equal to its peak cost.*

Lemma 6.3 (Reduction Cost Inclusion)**.** *If $\mathbb{c} \rightsquigarrow \mathbb{c}'$ then the peak cost of $\mathbb{c}'$ is less or equal to the peak cost of $\mathbb{c}$.*

It is important to observe that the proofs of Lemmas 6.2 and 6.3 are given using the (theoretical) solution of recurrence relations in [2]. This lets us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [10] or `PUBS` [2]. Given the basic cost and reduction cost inclusions, we can demonstrate the correctness theorem for our technique.

Theorem 6.4 (Correctness)**.** *Let $\overline{M}$ $\{\overline{F\,z\;\texttt{;}\;s'}\}$ be a well-typed program and let $\overline{\mathbb{C}}$, $\mathbb{c}$ be its behavioural type. Let also n be a solution of the function `translate`$(\overline{\mathbb{C}}, \mathbb{c})$. Then n is an upper bound of the number of virtual machines used during the execution of cn.*

The proof outline is as follows. Since the cost of the initial configuration cn is the direct cost of $\mathbb{c}$ then, by Lemma 6.2, this value is less or equal to the peak cost of $\mathbb{c}$. Let n be a solution of this cost. The argument proceeds by induction on the number of reduction steps:

- for the base case, when the program doesn't reduce, it turns out that $n \geq 1$;
- for the inductive case, let $cn \to cn'$. By applying Theorem 6.1 and Lemma 6.3, one derives that n is bigger than the peak cost of the behavioural type of cn'. Thus, by Lemma 6.2, we have that n is larger than the number of alive virtual machines in cn'.

7 Integration with a cost analysis tool and experiments

In this section we discuss technical details about the translation of the recurrence relations in Section 5 into the cost analysis tool we use – the `CoFloCo` analyser [10] – and we examine the outputs obtained for the running examples of this paper. It is worth to notice that instead of targeting `CoFloCo` , we might also target the `PUBS` analyser [2], which also has similar recurrence relations for input.

In order to comply with usual input formats of tools, we need to define encodings for vm values and for the functions `CNEW` and `CREL`. We therefore define

– $\top$ is modelled by `1`, ∂ is modelled by `2`, $\perp$ is modelled by `3`, and α by α. As regards $\alpha\downarrow$, it is modelled by the conditional value $[\alpha = \mathtt{3}]\mathtt{3}+[\mathtt{1} \leq \alpha \leq \mathtt{2}]\mathtt{2}$;

– the auxiliary functions `CNEW` and `CREL` are translated in recurrence relations as follows:

```
eq(CNEW(A), 0, [], [A = 3]).
eq(CNEW(A), 1, [], [A < 3]).

eq(CREL(A), -1, [], [A = 1]).
eq(CREL(A), 0, [], [A > 1]).
```

We begin our experiments with the translation of `double_release` when used by `user1`. In this case, the arguments of `double_release` are all different, therefore we use $\mathit{double\_release}_{\mathrm{peak}}(\alpha_1, \alpha_2, \alpha_3)$ and $\mathit{double\_release}_{\mathrm{net}}(\alpha_1, \alpha_2, \alpha_3)$. We write these functions in `CoFloCo` as `doubleRelease123_peak(A,B,C)` and `doubleRelease123_net(A,B,C)`. The input for the cost analyzer is shown in Figure 4 and, in order to evaluate it, we need to specify a so-called *entry point*. This entry point has the following format:

```
entry(METHOD_NAME (LIST_OF_ARGUMENTS):[CONDITIONS]).
```

where the first argument always represents the state of the carrier virtual machine. The following table report the output of `CoFloCo` with respect to the entry point. We observe that the computed cost is exactly what we anticipated

Entry Point	Cost
`entry(user1_net(1):[]).`	Maximum cost of `user1_net(1): 0`
`entry(user1_peak(1):[]).`	Maximum cost of `user1_peak(1): 2`

Table 1: Costs of programs `double_release` and `user1`

in Section 4.

Figure 5 describes the program of `double_release` when used by `user2`. In this case, the arguments of `double_release` are equal, therefore we use

```
eq(doubleRelease123_peak(A,B,C), 0, [], [A = 3]).
eq(doubleRelease123_peak(A,B,C), 0, [], [A < 3]).
eq(doubleRelease123_peak(A,B,C), 0, [crel(B)], [A < 3]).
eq(doubleRelease123_peak(A,B,C), 0, [crel(B), crel(C)], [A < 3]).

eq(doubleRelease123_net(A,B,C), 0, [], [A = 3]).
eq(doubleRelease123_net(A,B,C), 0, [doubleRelease123_peak(A,B,C)], [A = 2]).
eq(doubleRelease123_net(A,B,C), 0, [crel(B), crel(C)], [A = 1]).

eq(user1_peak(A), 0, [], [A = 3]).
eq(user1_peak(A), 0, [], [A < 3]).
eq(user1_peak(A), 0, [cnew(A)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A),doubleRelease123_peak(A, 1, 1)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A),doubleRelease123_net(A, 1, 1)], [A < 3]).

eq(user1_net(A), 0, [], [A = 3]).
eq(user1_net(A), 0, [user1_peak(A)], [A = 2]).
eq(user1_net(A), 0, [cnew(A),cnew(A),doubleRelease123_net(A, 1, 1)], [A = 1]).
```

Figure 4: Cost equations of `double_release` and `user1` in `CoFloCo` format

double_release$_{\text{peak}}(\alpha_1, \alpha_2)$ and *double_release*$_{\text{net}}(\alpha_1, \alpha_2)$. We write these functions in `CoFloCo` as `doubleRelease12_peak(A,B)` and `doubleRelease12_net(A,B)`.

```
eq(doubleRelease12_peak(A,B), 0, [], [A = 3]).
eq(doubleRelease12_peak(A,B), 0, [], [A < 3]).
eq(doubleRelease12_peak(A,B), 0, [crel(B)], [A < 3]).
eq(doubleRelease12_peak(A,B), 0, [crel(B), crel(3)], [A < 3]).

eq(doubleRelease12_net(A,B), 0, [], [A = 3]).
eq(doubleRelease12_net(A,B), 0, [doubleRelease12_peak(A,B)], [A = 2]).
eq(doubleRelease12_net(A,B), 0, [crel(B), crel(3)], [A = 1]).

eq(user2_peak(A), 0, [], [A = 3]).
eq(user2_peak(A), 0, [], [A < 3]).
eq(user2_peak(A), 0, [cnew(A)], [A < 3]).
eq(user2_peak(A), 0, [cnew(A),doubleRelease12_peak(A, 1)], [A < 3]).
eq(user2_peak(A), 0, [cnew(A),doubleRelease12_net(A, 1)], [A < 3]).

eq(user2_net(A), 0, [], [A = 3]).
eq(user2_net(A), 0, [user2_peak(A)], [A = 2]).
eq(user2_net(A), 0, [cnew(A),doubleRelease12_net(A, 1)], [A = 1]).
```

Figure 5: Cost equations of `double_release` and `user2` in `CoFloCo` format

The table below shows the output of the cost analyzer for the given equations, where, again, we consider only the case when the first argument is alive, that is, it is equal to 1. As before, the cost is exactly what we informally computed in Section 4.

Entry Point	Cost
`entry(user2_net(1):[]).`	0
`entry(user2_peak(1):[]).`	1

Table 2: Costs of programs `double_release` and `user2`

We conclude this section by discussing the cost of the two factorial programs `fact` and `cheap_fact` discussed in Section 2. The list of cost equations generated by `translate` is given in Figure 6 below.

```
eq(fact_peak(A,B), 0, [], [A = 3]).
eq(fact_peak(A,B), 0, [], [B = 0]).
eq(fact_peak(A,B), 0, [], [B > 0]).
eq(fact_peak(A,B), 0, [cnew(A)], [B > 0]).
eq(fact_peak(A,B), 0, [cnew(A), fact_peak(1, B-1)], [B > 0]).
eq(fact_peak(A,B), 0, [cnew(A), fact_net(1, B-1)], [B > 0]).
eq(fact_peak(A,B), 0, [cnew(A), fact_net(1, B-1), crel(1)], [B > 0]).

eq(fact_net(A,B), 0, [], [A = 3]).
// eq(fact_net(A,B), 0, [fact_peak(A,B)], [A = 2]).
eq(fact_net(A,B), 0, [cnew(A), fact_net(1, B-1), crel(1)], [A = 1, B > 0]).
eq(fact_net(A,B), 0, [], [A = 1, B = 0]).


eq(cheap_fact_peak(A,B,C), 0, [], [A = 3]).
eq(cheap_fact_peak(A,B,C), 0, [], [B = 0]).
eq(cheap_fact_peak(A,B,C), 0, [], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A)], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A), prod_peak(1,B,C)], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A), prod_net(1,B,C)], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A), prod_net(1,B,C), crel(1)], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A), prod_net(1,B,C), crel(1),
                                    cheap_fact_peak(1, B-1, D)], [B > 0]).
eq(cheap_fact_peak(A,B,C), 0, [cnew(A), prod_net(1,B,C), crel(1),
                                    cheap_fact_net(1, B-1, D)], [B > 0]).

eq(cheap_fact_net(A,B,C), 0, [], [A = 3]).
//eq(cheap_fact_net(A,B,C), 0, [cheap_fact_peak(A,B,C)], [A = 2]).
eq(cheap_fact_net(A,B,C), 0, [cnew(A), prod_net(1,B,C), crel(1),
                                    cheap_fact_net(1, B-1, D)], [A = 1, B > 0]).
eq(cheap_fact_net(A,B,C), 0, [], [A = 1, B = 0]).

eq(prod_peak(A,B,C),0,[],[]).
eq(prod_net(A,B,C),0,[],[]).
```

Figure 6: Cost equations of `fact` and `cheap_fact` in `CoFloCo` format

We notice that Figure 6 has a couple of equations commented out. These equations are obtained by the `translate` function in Section5. However it is not allowed as input in `CoFloCo` because those equations lead to mutually recursive chains, which is banned by this tool. Yet, in this case, the exclusion of this equation does not affect the result because we assume that the virtual machine executing either `fact` or `cheap_fact` is always alive. Table 3 shows the output of `CoFloCo`.

Entry Point	Cost
`entry(fact_net(1,B):[B>=0]).`	`0`
`entry(fact_peak(1,B):[B>=0]).`	`max([B,1])`
`entry(cheap_fact_net(1,B):[B>=0]).`	`0`
`entry(cheap_fact_peak(1,B):[B>=0]).`	`1`

Table 3: Costs of programms `fact` and `cheap_fact`

As in the previous examples the *net cost* is equal to `0` in both cases, bacause every created virtual machine is released before the end of the program. In the case of the *peak cost*, for method `fact` the number of virtual machines will depend on the depth of the recursion, in this case, given by parameter `B`. On the other hand for method `cheap_fact` at each step the created VM is released before the recursive call producing a *peak cost* equal to 1.

8 Related Work

After the pioneering work by Wegbreit in 1975 [20] that developed a technique for deriving upper-bounds costs of functional programs, a number of techniques for the cost analysis have been developed. These techniques may be divided into two categories.

The first category, that we call *classical techniques*, addresses cost analysis in three steps: *(i)* extracting relevant informations out of the original programs, *e.g.* abstracting data structures to their size and assigning a cost to eevery program expression, *(ii)* converting the abstract program into recurrence relations, and *(iii)* solving the cost equations with an automatic tool. Very powerful classical technique tools are [2, 12, 3, 9, 10] that produce very accurate upper bounds expressions for various kinds of programs of different complexity. We refer to [10] for a comparison among some of these techniques. The main drawback of these techniques is the lack of compositionality: it is difficult to scale the analysis to large programs and the translation from the original program to the recurrence relations is always unclear.

The second category, that we call *amortized techniques*, uses the technique based on type systems and amortized analysis developed for functional programs by Hofmann [15]. This approach is highly compositional because of the use of types, and more suitable for formal verification because the connection between the original program and the cost equations can be demonstrated by a standard subject-reduction theorem [15, 19, 16, 17, 13]. We follow this technique in the present work.

A common feature of classical and amortized techniques is that they analyze *cumulative resources*, that is resources that *do not decrease* during the execution of the programs. This is the case, for example, for execution time, number of operations, memory (without an explicit `free` operation), etc. As we discussed, this assumption eases the analysis because it permits to compute over-approximated cost. On the contrary, the presence of *explicit or implicit* release operation entangles the analysis, as already discussed in [6] where a memory cost analysis is proposed for languages with garbage collection. It is worth to say that the scenario of [6] is not difficult because, by definition of garbage collection, released memories are always inactive. The impact of the release operation in the cost analysis is thoroughly discussed in [7] by means of the notions of *peak cost* and *net cost*. As discussed in Section 5, the first refers to the worst case cost for an operation to complete, while the second refers to the cost of an operation after completion. For cumulative analysis the two notions coincide; however, in non-cumulative analysis (in presence of a release operation) they are different and the *net cost* is key for computing tight upper bounds.

Recently Albert *et al.* in [4] have analysed the cost of a language with explicit releases. However, the release operation they consider is used in a very restrictive way: releases can only be performed over locally acquired resources. This constraint ensures having no partially negative costs when analyzing block sequences thus maintaining the restriction of non-negative cost models.

Most cost analysis techniques usually address sequential program. Only few works also address concurrent programs [1, 5, 14]. In this cases, to reduce the imprecision of the analysis caused by the nondeterminism, the authors of [1, 5] use a clever technique for determining parallel codes, called *may-happen-in-parallel* [8]. However no one of these contributions consider a concurrent language with a powerful `release` operation that allows one disable the resources taken in input. In facts, without this operation, one can model the cost by simply aggregating the sets of operations that can occur in parallel, as in [5], and all the theoretical development is much easier.

9 Conclusion

This paper presents the first (to the best of our knowledge) static analysis technique that computes upper bound of virtual machines usages in concurrent programs that may create and, more importantly, may release such machines. Our analysis consists of a type system that extracts relevant informations about resource usages in programs, called behavioural types; an automatic translation that transforms these types into cost expressions; the application of solvers, like `CoFloCo` [10], on these expressions that compute upper bounds of the usage of virtual machines in the original program. A relevant property of our technique is its modularity. For the sake of simplicity, we have applied the technique to a small language. However, by either extending or changing the type system, the analysis can be applied to many other languages with primitives for creating and releasing resources. In addition, by changing the translation algorithm, it is possible to target other solvers that may compute better upper bounds.

For the future, we consider three lines of work. First, we will complete the technical development of this paper by delivering full proofs of correctness of our technique. Second, we intend to alleviate the restrictions introduced in Section 3 on the programs we can analyse. This may be pursued by retaining more expressive notations for the effect of a method, *i.e.* by considering `R` as a set of sets instead of a simple set. Such a notation is more suited for modelling nondeterministic behaviours and it might be made even more expressive by tagging all the different effects in `R` with a condition specifying when such effect is yielded. Third, we intend to implement our analysis targeting a programming language with a formal model as `ABS` [18].

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla. Cost analysis of concurrent OO programs. In *Proceedings of Programming Languages and Systems - APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings SAS 2008*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[3] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[4] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of (TACAS 2015)*, Lecture Notes in Computer Science. Springer, 2015. To appear.

[5] Elvira Albert, Jess Correas, and Guillermo Romn-Dez. Peak cost analysis of distributed systems. In *Proceedings of SAS 2014*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014.

[6] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. *SIGPLAN Not.*, 45(8):121–130, 2010.

[7] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis*, pages 405–421. Springer, 2012.

[8] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *Proceedings of LCPC 2005*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2006.

[9] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer

programs. In *Proceedings of TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[10] Antonio Flores Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems, Singapore*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, November 2014.

[11] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2015. To Appear.

[12] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[13] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, volume 46, pages 357–370. ACM, 2011.

[14] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs, 2015. [Online; accessed 11-February-2015].

[15] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, volume 38, pages 185–197. ACM, 2003.

[16] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *Programming Languages and Systems*, pages 22–37. Springer, 2006.

[17] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Computer Science Logic*, pages 317–331. Springer, 2009.

[18] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[19] Wei ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer, 2005.

[20] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.