Static analysis of cloud elasticity

Abel Garcia, Cosimo Laneve, Michael Lienhardt

Static analysis of cloud elasticity *

Abel Garcia Cosimo Laneve Michael Lienhardt
Department of Computer Science and Engineering, University of Bologna – INRIA Focus
{abel.garcia2, cosimo.laneve, michael.lienhardt}@unibo.it

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) concurrent code 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 information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioural types and return cost equations, and (*iii*) an automatic off-the-shelf solver for the cost equations. A soundness proof of the type system establishes the correctness of our technique with respect to the cost equations. We have experimentally evaluated our technique using a cost analysis solver and we report some results. The experiments show that our analysis allows us to derive bounds for programs that are better than other techniques, such as those based on amortized analysis.

Categories and Subject Descriptors F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages—Operational semantics,Program analysis ; F.1.1 [*Computation by abstract devices*]: Models of Computation—Relations between models

General Terms Static analysis, Resource consumption, Concurrent programming, Behavioural type system, Subject reduction.

Keywords Virtual machines creations and releases, transition relation, behavioural types, peak cost, net cost, cost equations.

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 fact, 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 simple code examples, extrapolating this information for fully real-life complex programs could be cumbersome and highly error-sensitive. The first attempts about the analysis of resource usage dates back to Wegbreit's pioneering work in 1975 [21], 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 [4, 9]) and to make the connection between programs and closed-form expressions precise (see for instance [10, 15]). A more detailed discussion of the related work in the literature is presented in Section 7.

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 [6]. 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? A solution to this problem, under the assumption that one can acquire a virtual machine that has been previously released, 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:

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

– 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 a precise cost of methods like `double_release`, in Section 4 we associate methods with abstract descriptions that carry information about resource usages. These descriptions are called *behavioural types* and are formally connected to the programs by means of a type system.

The analysis of behavioural type is defined in Section 5 by translating them in a code that is adequate for an off-the-shelf solver – the `CoFloCo` solver [11]. 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 fact, 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 two *equal* arguments then its cost cannot be computed by a 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. The translation of behavioural types into `CoFloCo` input code has been prototyped and we are therefore able to automatically compute the cost of programs. In Section **??** we examine our prototype implementation and we report the results of some of our experiments. It is worth to notice that our technique (and, consequently, our prototype) allows us to derive bounds for programs that are better than other techniques, such as those based on amortized analysis. We address this topic in Section 7.

Our technique targets a simple concurrent language with explicit operations of creation and release of resources. The language is defined in Section 2 and we discuss restrictions that ease the development of our technique in Section 3. In Section 6 we outline our correctness proof of the type system with respect to the cost equations. Due to page constraints, the details of the proof are omitted and appear in the full paper. In Section 8 we deliver concluding remarks.

In this paper we use the metaphor of cloud computing and virtual machines. We observe that our technique may be also used for resource analysis of concurrent languages that bear operations of acquire (or creation) and release (such as heaps).

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<`T`>`. 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:

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

A statement s may be either one of the standard operations of an imperative language plus the `release` x operation which marks the virtual machine x for disposal.

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 *non blocking* `get` operation: if the value needed by a process is not available then an awaiting process is scheduled and executed. Expressions z also include `new Vm` that creates a new virtual machine. The intended meaning of operations taking place on different virtual machines is that they may execute in parallel, while operations in the same virtual machine interleave their evaluation (even if in the following operational semantics the parallelism is not explicit). The execution of method invocations and creations and releases of machines always returns an erroneous value when executed on a released machine.

A (*pure*) expression e are the reserved identifier `this`, the virtual machines identifiers and the integer expressions. Since our analysis will be parametric with respect to the inputs, we parse integer expressions in a careful way. In particular we split them into *size expressions* se, which are expressions in Presburger arithmetics (this is a decidable fragment of Peano arithmetics that only contains addition), and *non-size expressions* nse, which are the other type of expressions. The syntax of size and non-size expressions is the following:

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

In the whole paper, we assume that sequences of declarations $\overline{T\,x}$ and method declarations $\overline{M}$ do not contain duplicate names. We also assume that `return` statements have no continuation.

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

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

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', $\cdots$. 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* $\mathit{vm}(o,a,p,q)$ where o is a vm name; a is either $\top$ or $\bot$ according to the machine is alive or dead, p

is either $\{l \mid \epsilon\}$, 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 $\perp$ then the actual value of f has still to be computed.

– *method invocation messages* $invoc(o, f, \mathtt{m}, \overline{v})$.

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 in l. As regards boolean operations, as usual, `false` is represented by `0` and `true` is represented by a value different from `0`. 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 $[\![f]\!]_l = f$. It is worth to notice that $[\![e]\!]_l$ is undefined whenever e contains a variable that is not defined in l.

– $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}) = \{[\overline{x} \mapsto \overline{v},\ \mathrm{destiny} \mapsto f] \mid s[^o/_{\mathtt{this}}]\}$, 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 the active process p. The rules are almost standard, except those about the management of virtual machines and the method invocation, which we are going to discuss.

(NEW-VM) creates a virtual machine and makes it active – rule (NEW-VM). 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 $\perp$ – rules (RELEASE-VM) and (RELEASE-VM-SELF). If instead the virtual machine executing the `release` has been already released, then the operation has no effect – rule (RELEASE-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 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

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

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 ;
  if (n==0) { return 1 ; }
  else { x = this!fact(n-1) ; m = x.get ;
         return m*n ; }
}
Int costly_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 m*n; }
}
```

The method `fact` is the standard definition of factorial with the recursive invocation `fact(n-1)` always performed on the same machine. That is, to compute `fact(n)` one needs one virtual machine. On the contrary, the method `costly_fact` performs the recursive invocation 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 `m*n`. Notice that every vm creation occurs before any release operation. As a consequence, `costly_fact` will create as many virtual machines as the argument n. That is, if the application has only k virtual machines then `costly_fact` cab compute factorials up-to $k-1$ (1 is the virtual machine executing the method).

The analysis of `costly_fact` has been easy because the `release` operation is applied on a locally created virtual machine. Yet, in `vml`, `release` may also apply to method arguments and the presence of this feature in concurrent codes is the major source of difficulties for the analysis. A paradigmatic example is the `double_release` method discussed in Section 1 that may have either a cost of `-2` or of `-1` or of `0`. It is worth to observe that, while over-approximations (*e.g* not counting releases) return (too) imprecise costs, under-approximations may return wrong costs. For example, the following method

```
Int fake_method(Int n) {
   if (n=0) return 0 ;
   else { Vm x ; Fut<Int> f ;
          x = new Vm ; x = new Vm ;
          f = this!double_release(x,x) ; f.get ;
          f = this!fake_method(n-1) ; f.get ;
          return 0 ; }
}
```

creates two virtual machines and releases the second one with `this!double_release(x,x)` before the recursive invocation. We notice that `fake_method(n)` should have cost `n`. However

– an under-approximation of `double_release` (cost `-2`) gives `0` as cost of `fake_method(n)`.

The aim of the following sections is to define a technique for determining the cost of method invocations that makes these costs depend on the identity and on the state of method's arguments, as well as on those arguments that are released.

3. Determinacy of releases of method's arguments

Our cost analysis of virtual machines uses abstract descriptions that carry informations about method invocations and creations and removals of virtual machines. In order to ease the compositional reasonings, method's descriptions also defines the arguments the method releases upon termination. In this contribution we stick to method descriptions that are as simple as possible, namely we assume that the arguments a method releases upon termination are *a set*. In turn, this requires that methods' behaviours are *deterministic* with respect to such releases. To enforce this determinacy, we constrain the language `vml` as follows.

Restriction 1: *the branches in a method body always release the same set of method's arguments.*

For example, methods like

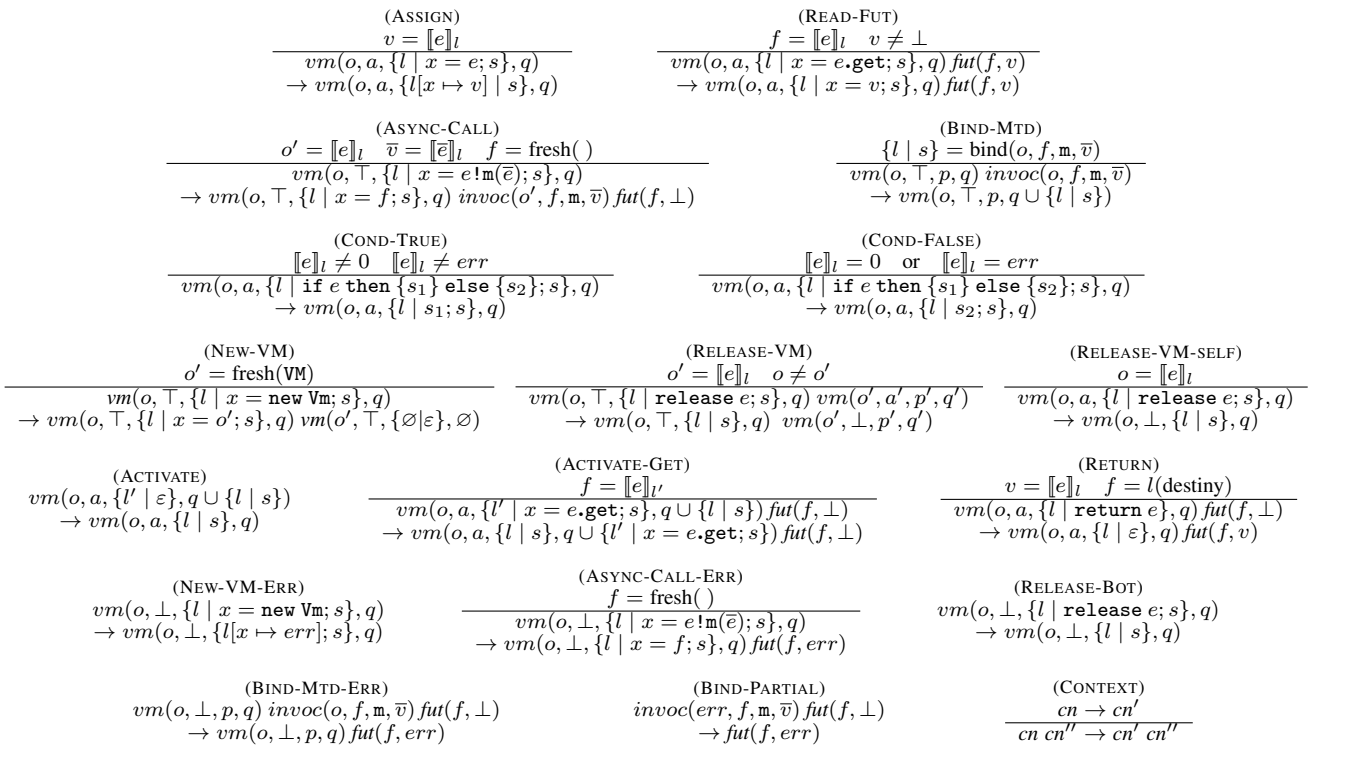

Figure 1. Semantics of `vml`.

```
Int foo1(Vm x, Int n) {
    if (n = 0) return 0 ;
    else { release x ; return 0; }
}
```

cannot be handled by our analysis because the else-branch releases the argument `x` while the then-branch does not release anything.

Restriction 2: *method invocations are always synchronized within caller's body.* In this way every effect of a method is computed before its termination. For example, methods like

```
Int foo2(Vm x, Vm y) {
    this!double_release(x,y) ; return 0 ;
}
```

cannot be handled by our analysis because it is not possible to determine that the arguments `x` and `y` of `foo2` will be released or not upon its termination because the invocation to `double_release` is asynchronous.

Restriction 3: *machines that are executing methods that release arguments must be alive*. (This includes the carrier machine, *e.g.* method bodies cannot release the `this` machine.) Here (we are at static time) "alive" means that the machine is either the caller or has been locally created and has not been/being released. For example, in `foo3`

```
Int simple_release(Vm x) { release x; return 0; }
Int foo3(Vm x) {
    Vm z ; Fut<Int> f ;
    z = new Vm ; f = z!simple_release(x) ;
    release z ; f.get ; return 0;
}
```

the machine `z` is released before the synchronisation with the `simple_release` – statement `f.get`. This means that the disposal of `x` depends on scheduler's choice, which means that it is not possible to determine whether `foo3` will release `x` or not. A similar issue arises when the callee of a method releasing arguments is itself an argument. For example, in `foo4`

```
Int foo4(Vm x, Vm y) {
    Fut<Int> f ;
    f = x!simple_release(y) ;
    f.get ; return 0 ;
}
```

it is not possible to determine whether `y` is released or not because the state of `x` cannot be determined.

Restriction 4: *if a method returns a machine, the machine must be new.* For example, consider the following code:

```
Vm identity(Vm x) { return x; }
{
   Vm x ; Vm y ; Vm z ; Fut<Vm> f ; Fut<Int> g ; Int m ;
   x = new Vm ; y = new Vm ;
   f = x!identity(y) ; g = this!simple_release(x);
   z = f.get ; m = g.get ;
   release z ;
}
```

In this case it is not possible to determine whether the value of `z` is `x` or `err` and, therefore, it is not clear whether the cost of `release z` is 0 or `-1`. The problem is `identity`, which returns the argument that is going to be released by a parallel method. The Restriction 4 bans methods like `identity` because it does not return a fresh machine. In fact, such machines cannot be released by a parallel method.

Restrictions 1, 3, and 4 are enforced by the type system in Section 4, in particular by rules (T-METHOD), (T-INVOKE) and (T-RELEASE), and (T-INVOKE) and (T-RETURN), respectively. Restriction 2 is a programming constraint; it may be released by using a continuation passing style that entangles a lot both the type system and the analysis (see [12] for a possible solution that has been designed for deadlock analysis).

4. The behavioural type system of vml

Behavioural types are abstract codes highlighting the features of vml programs that are relevant for the cost analysis in Section 5. These types support compositional reasonings and are associated to programs by means of a type system that is defined in this section.

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 presented in Figure 2.

Behavioural types express creations of virtual machines ($\nu\alpha$) and their removal ($\alpha^{\checkmark}$), method invocations ($\nu f : \mathtt{m}\,\alpha(\overline{\mathtt{s}}) \to \mathtt{o}$) and the corresponding retrieval of the value ($f^{\checkmark}$), and the conditionals (respectively $(se)\{\mathbb{c}\} + (\neg se)\{\mathbb{c}'\}$ or $\mathbb{c} + \mathbb{c}'$, according to whether the boolean guard is a size expressions that depends on the arguments of a method or not). We will always shorten the type $\nu f : \mathtt{m}\,\alpha(\overline{\mathtt{s}}) \to \mathtt{o}$ into $\nu f : \mathtt{m}\,\alpha(\overline{\mathtt{s}})$ whenever $\mathtt{o} = \_$.

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 $\alpha(\overline{\mathtt{r}}) : \mathtt{o}, \mathtt{R}$, from variables to extended values $\mathbb{x}$, from future names to future values, and from vm names to extended values $\mathtt{F}\mathbb{t}$, which are called *vm states* in the following. These environments occurring in the leaves are only used in the typing proofs and are dropped in the final types (method types and the main statement type).

Vm states $\mathtt{F}\mathbb{t}$ are a collection of future names $\mathtt{F}$ plus the value $\mathbb{t}$ of the virtual machines. This $\mathtt{F}$ specifies the set of parallel methods that are going to release the virtual machine; $\mathbb{t}$ defines whether the virtual machine is alive $\top$, or it has been already released ($\bot$) or, according to scheduler's choices, it may be either alive or released (∂). Vm values also include terms α and $\alpha\!\downarrow$. The value α is given to the argument machines of methods (they will be instantiated by the invocations – see the cost analysis in Section 5), the value $\alpha\!\downarrow$ is given to argument values that are returned by methods *and* can be released by parallel methods ($\alpha\!\downarrow$ will be also evaluated in the cost analysis). Vm values are partially ordered by the relation $\leq$ defined by

$$\partial \leq \top \qquad \partial \leq \bot \qquad \alpha\!\downarrow \leq \bot \qquad \alpha\!\downarrow \leq \alpha\ .$$

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 $\partial \sqcap \alpha\!\downarrow$ is not defined.

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 $\Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathtt{r}}) : \mathtt{o}, \mathtt{R}$ 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 $x \notin \mathrm{dom}(\Gamma)$ and the environment update

$$\Gamma[x \mapsto \mathbb{x}](y) \stackrel{def}{=} \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

With an abuse of notation (see rule (T-RETURN)), we let $\Gamma[\_ \mapsto \mathbb{x}] \stackrel{def}{=} \Gamma$ (because $\_$ does not belong to any environment). We will also use the operation and notation below:

- $\mathtt{F}\mathbb{t}\Downarrow$ is defined as follows:

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

  and we write $(\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n)\Downarrow$ for $(\mathtt{F}_1\mathbb{t}_1\Downarrow, \cdots, \mathtt{F}_n\mathbb{t}_n\Downarrow)$.
- the *multihole contexts* $\mathcal{C}[\,]$ defined by the following syntax:

$$\mathcal{C}[\,] ::= \quad [\,] \quad | \quad \mathtt{a} \,;\, \mathcal{C}[\,] \quad | \quad \mathcal{C}[\,] + \mathcal{C}[\,] \quad | \quad (se)\{\mathcal{C}[\,]\}$$

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

The type system for expressions is reported in Figure 3. It is worth to notice that this type system 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.

The type system for expressions with side effects and statements is reported in Figure 4. We discuss rules (T-INVOKE), (T-GET), (T-RELEASE), and (T-NEW).

Rule (T-INVOKE) types method invocations $e!\mathtt{m}(\overline{e})$ by using a fresh 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. This value contains the returned value, the vm name of the callee and its state, and the set of the arguments that the method is going to remove. The vm state of the callee will be used when the method is synchronized to update the state of the returned object, if any (see rule (T-GET)). It is important to observe that the environment returned by (T-INVOKE) is updated with information 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 α': 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 subcases, 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, as we discussed in Restriction 3 in Section 3. We observe that, 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. This constraint enforces Restriction 3 in Section 3. The constraint in the third line of the premise of (T-INVOKE) enforces Restriction 3 to the other invocations in parallel and to the object executing $e!\mathtt{m}(\overline{e})$.

Rule (T-GET) defines the synchronisation with a method invocation that corresponds to a future f. Let $(\mathtt{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})$ be the value of f in the environment. Since $\mathtt{R}$ 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 the returned value is a virtual machine that has been created by the method of f, its state is the same of the callee vm name (which may have been updated since the invocation), namely the value of $\mathbb{t} \sqcap (\Gamma(\alpha)\Downarrow)$.

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

Figure 2. Behavioural Types Syntax

$$
\text{(T-VAR)}\ \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad
\text{(T-PRIMITIVE)}\ \Gamma \vdash \mathtt{p} : \mathtt{p} \qquad
\text{(T-OP)}\ \frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1\ \mathtt{op}'\ e_2 : se_1\ \mathtt{op}'\ se_2}
$$

$$
\text{T-UNIT}\ \frac{\Gamma \vdash e : se}{\Gamma \vdash e : \_} \qquad
\text{(T-OP-UNIT)}\ \frac{\Gamma \vdash e_1 : \_\ \ \mathtt{or}\ \ \Gamma \vdash e_2 : \_\ \ \mathtt{or}\ \ \mathtt{op} \in \{*, /\}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : \_}
$$

$$
\text{(T-PURE)}\ \frac{\Gamma \vdash e : \mathbb{x}}{\Gamma \vdash e : \mathbb{x}, \mathtt{0} \triangleright \Gamma} \qquad
\text{(T-METHOD-SIG)}\ \frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R} \quad \overline{\beta} \subseteq \mathit{fv}(\alpha, \overline{\mathbb{r}}, \mathbb{o}) \\ \sigma \text{ is a vm renaming such that } \mathbb{o} \notin \mathit{fv}(\alpha, \overline{\mathbb{r}}) \text{ implies } \sigma(\mathbb{o}) \text{ fresh}\end{array}}{\Gamma \vdash \mathtt{m}\,\sigma(\alpha)(\sigma(\overline{\mathbb{r}})) : \sigma(\mathbb{o}), \sigma(\mathtt{R})}
$$

Figure 3. Typing rules for expressions

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

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

$$
\text{(T-INVOKE-BOT)}\ \frac{\Gamma \vdash e : \alpha \qquad \Gamma(\alpha) = \mathtt{F}\bot \qquad f \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \mathtt{0} \triangleright \Gamma'[f \mapsto \_]}
$$

$$
\text{(T-GET)}\ \frac{\begin{array}{c}\Gamma \vdash x : f \qquad \Gamma \vdash f : (\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R}) \\ \mathtt{R}' = \mathit{fv}(\mathbb{o}) \setminus \mathtt{R} \qquad \mathbb{t}' = \mathbb{t} \sqcap (\Gamma(\alpha)\Downarrow) \\ \Gamma' = \Gamma[\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}'}\end{array}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, f^{\checkmark} \triangleright \Gamma'[f \mapsto \mathbb{o}]}
$$

$$
\text{(T-GET-DONE)}\ \frac{\Gamma \vdash x : f \qquad \Gamma \vdash f : \mathbb{o}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, \mathtt{0} \triangleright \Gamma}
$$

$$
\text{(T-NEW)}\ \frac{\beta \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} \mathtt{new\ Vm} : \beta, \nu\beta \triangleright \Gamma[\beta \mapsto \varnothing\top]} \qquad
\text{(T-RELEASE)}\ \frac{\begin{array}{c}\Gamma \vdash x : \alpha \\ \alpha \notin \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \mathbb{x}, \mathtt{R}') \text{ and } \mathtt{R}' \neq \varnothing\}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}\ x : \alpha^{\checkmark} \triangleright \Gamma[\alpha \mapsto \varnothing\bot]}
$$

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

$$
\text{(T-SEQ)}\ \frac{\Gamma \vdash_{\mathtt{S}} s_1 : \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n] \qquad (\Gamma_i \vdash_{\mathtt{S}} s_2 : \mathbb{c}'_i)^{i \in 1..n}}{\Gamma \vdash_{\mathtt{S}} s_1; s_2 : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{c}'_1] \cdots [\mathbb{a}_n \fatsemi \mathbb{c}'_n]} \qquad
\text{(T-RETURN)}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathbb{o} \qquad \Gamma \vdash \mathtt{destiny} : \mathbb{o}' \\ \mathbb{o} \notin \mathtt{S}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{return}\ e : \mathtt{0} \triangleright \Gamma[\mathbb{o}' \mapsto \Gamma(\mathbb{o})]}
$$

Figure 4. Type rules for expressions with side effects and statements.

Rule (T-RELEASE) models the removal of a vm name α. The premise in the second line verifies that the disposal do not address machines that are executing methods, as discussed in Restriction 3 of Section 3.

The type system of `vml` is completed with the rules for method declarations and programs, given in Figure 5.

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 fresh vm name ($\mathbb{o} \notin \{\alpha\} \cup \overline{\beta}$) as discussed in Restriction 4 of Section 3. The constraints in the third line of the premises of (T-METHOD) implement Restriction 1 of Section 3. We also observe that $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup \mathit{fv}(\mathbb{o})}$ guarantees that every branch of the be-

(T-METHOD)

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

(T-PROGRAM)

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

Figure 5. Behavioural typing rules of method and programs.

havioural type creates a new vm name and, by rule (T-RETURN), the state of the chosen vm name must be always the same.

We display behavioural types examples by using codes from Sections 1 and 2. Actually, the following types do not abstract a lot from codes because the programs of the previous sections have been designed for highlighting the issues of our technique.

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

```
fact α(n) {
 (n==0){ 0 }
 +(n>0){ ν y:fact α(n − 1) ; y✓ }
} - , { }
```

```
costly_fact α(n) {
 (n==0){ 0 }
 +(n>0){ νβ ;
       ν x : costly_fact β(n − 1) ;
       x✓ ; β✓ }
} - , { }
```

and it is worth to highlight that the type of `costly_fact` records the order between the recursive invocation and the release of the machine.

The behavioural type of `double_release` is the following one

```
double_release α(β, γ) {β✓ ; γ✓ } - , {β, γ}
```

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.

5. The analysis of behavioural types

The types returned by the system in Section 4 are used to compute the resource cost of a `vml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [11] – that takes in input a set of so-called *cost equations*. `CoFloCo` cost equations are terms

$$m(\overline{x}) = exp \quad [se]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbols applications (we do not define the syntax of *exp*, which may be derived by the following equations; the reader is referred to [11]), and *se* is a size expression whose variables are contained in $\overline{x}$.

Basically, our translation maps method types into cost equations, where

- method invocations are translated into function applications,
- 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 $\mathtt{m}_{\mathrm{peak}}$, and the other returns the number of resources the method `m` creates without releasing, called *net cost* of `m` and noted $\mathtt{m}_{\mathrm{net}}$. 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 `m'` is the maximal value between $\mathtt{m}_{\mathrm{peak}}$, $\mathtt{m}_{\mathrm{net}} + \mathtt{m}'_{\mathrm{peak}}$, and $\mathtt{m}_{\mathrm{net}} + \mathtt{m}'_{\mathrm{net}}$; while the cost of the parallel execution of `m` and `m'` is the maximal value between $\mathtt{m}_{\mathrm{peak}} + \mathtt{m}'_{\mathrm{peak}}$, $\mathtt{m}_{\mathrm{net}} + \mathtt{m}'_{\mathrm{peak}}$, $\mathtt{m}_{\mathrm{net}} + \mathtt{m}'_{\mathrm{peak}}$, and $\mathtt{m}_{\mathrm{net}} + \mathtt{m}'_{\mathrm{net}}$.

There are two difficulties that entangle our translation, both related to method invocations: the management of arguments' identities and of arguments' values.

Arguments' identities. Consider the code

```
Int simple_release(Vm x) { release x ; return 0 ; }

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

The behavioural types of these methods are

```
simple_release α(β){ β✓ } ⊥, {β}
m α(β, γ){ν f : simple_release α(β) ; γ✓ ; f✓ } ⊥, {β, γ}
```

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 fact, 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 $\mathtt{free}(\beta)$ and therefore it counts `0`. To solve this problem of arguments' identity, we refine even more the translation of a method type, which now depends on an equivalence relation telling which of the vm names in parameter are actually equal or not. Hence, the above method `m` is translated in four cost functions: $\mathtt{m}_{\mathrm{peak}}^{\{1\},\{2\}}(x,y)$ and $\mathtt{m}_{\mathrm{net}}^{\{1\},\{2\}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\mathtt{m}_{\mathrm{peak}}^{\{1,2\}}(x)$ and $\mathtt{m}_{\mathrm{net}}^{\{1,2\}}(x)$, which correspond to the invocations where $x = y$. (The equivalence relation in the superscript never mention `this`, which is also an argument, because, in this case `this` cannot be identified with 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:

$$\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n) \ = \bigcup_{i \in 0..n} \{\ \{j \mid \alpha_j = \alpha_i\}\ \}$$

Let $\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n)(\beta_0, \cdots, \beta_n)$ be the tuple $(\beta_{i_1}, \cdots, \beta_{i_k})$, where $i_1, \cdots, i_k$ are *canonical representatives* of the sets in

$\texttt{EqRel}(\alpha_0, \cdots, \alpha_n)$ (we take the vm name with the least index in every set). We observe that, by definition, $\texttt{EqRel}(\alpha_0, \cdots, \alpha_n)(\alpha_0, \cdots, \alpha_n)$ is a tuple of pairwise different vm names (in $\alpha_0, \cdots, \alpha_n$).

Without loosing in generality, we will always assume that the canonical representative of a set containing 0 is always 0. This index represents the `this` object and we remind that, by Restriction 3 in Section 3, such an object cannot be released. This is the reason why, in the foregoing discussion about the method `m`, we did not mentioned `this`. Additionally, in order to simplify the translation of method invocations, we also assume that the argument `this` is always different from other arguments (the general case just requires more details).

(Re)computing argument's states. In Section 4 we computed the state of every machine in order to enforce the restrictions in Section 3. In this section we mostly compute them again for a different reason: obtaining a (more) precise cost analysis. Of course one might record the computation of vm states in behavioural types. However, this solution has the drawback that behavioural types become unintelligible because they carry informations that is needed by the analyser.

Let a *translation environment*, ranged over Ψ, Ψ', be a mapping from vm names to vm states and from future names to triples $(\Psi', \texttt{R}, \texttt{m}\ \beta(\overline{se}, \overline{\beta}) \to \mathfrak{o})$, where Ψ' is a translation environment defined on vm names only, called *vm-translation environment*. We define the following auxiliary functions

– let Ψ be a vm-translation environment. Then

$$\Psi|_X(\alpha) \stackrel{def}{=} \begin{cases} \Psi(\alpha) & \text{if } \alpha \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

– the *update of a vm-translation environment* Ψ with respect to f and Ψ', written $\Psi \searrow^f_\downarrow \Psi'$, returns a vm-translation environment defined as follows:

$$(\Psi \searrow^f_\downarrow \Psi')(\alpha) \stackrel{def}{=} \begin{cases} (\texttt{F}' \setminus \{f\})(\mathbb{t} \sqcap \mathbb{t}') & \text{if } \Psi(\alpha) = \texttt{F}\mathbb{t} \\ & \text{and } \Psi'(\alpha) = \texttt{F}'\mathbb{t}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

This operation $\Psi \searrow^f_\downarrow \Psi'$ updates the vm-translation environment Ψ of a method invocation, which is stored in the future f, with respect to the translation environment at the synchronisation point. It is worth to observe that, by definition of our type system and the following translation function, the values of $\Psi(\alpha)$ and $\Psi'(\alpha)$ are related. In particular, if $\mathbb{t} = \alpha$ then $\mathbb{t}'$ can be either α or $\alpha\downarrow$ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation on the future f); if $\mathbb{t} = \top$ then $\mathbb{t}'$ can be either $\top$ or ∂ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation).

– the *merge operation*, noted $\Psi(\Delta)$, where Ψ is a vm-translation environment and Δ is an equivalence relation, returns a *substitution* defined as follows. Let

$$\mathbb{t} \otimes^\alpha \mathbb{t}' \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathbb{t} = \bot \text{ or } \mathbb{t}' = \bot \\ \alpha & \text{if } \mathbb{t} \text{ and } \mathbb{t}' \text{ are variables} \\ \alpha\downarrow & \text{otherwise} \end{cases}$$

$$\texttt{F}_1\mathbb{t}_1 \otimes^\alpha \texttt{F}_2\mathbb{t}_2 \stackrel{def}{=} (\texttt{F}_1 \cup \texttt{F}_2)(\mathbb{t}_1 \otimes^\alpha \mathbb{t}_2)$$

Then

$$\Psi(\Delta) \ : \ \alpha \mapsto \bigotimes^{\Delta(\alpha)} \{\Psi(\beta) \mid \beta \in \text{dom}(\Psi) \text{ and } \Delta(\beta) = \Delta(\alpha)\}$$

for every $\alpha \in \text{dom}(\Psi)$.

The operator $\otimes^\alpha$ has not been defined on vm values as ∂ or $\top$ because we merge vm names whose image by Ψ can be either $\texttt{F}\beta$ or $\texttt{F}\beta\downarrow$ or $\texttt{F}\bot$. As a notational remark, we observe that $\Psi(\Delta)$ is noted as a map $[\alpha_1 \mapsto \texttt{F}_1\mathbb{t}_1, \cdots, \alpha_n \mapsto \texttt{F}_n\mathbb{t}_n]$ instead of the standard notation $[\texttt{F}_1\mathbb{t}_1, \cdots, \texttt{F}_n\mathbb{t}_n/_{\alpha_1, \cdots, \alpha_n}]$. These two notations are clearly equivalent: we prefer the former one because it will let us to write $\Psi(\Delta)(\alpha)$ or even $\Psi(\Delta)(\alpha_1, \cdots, \alpha_n)$ with the obvious meanings.

To clarify the reason for a merge operator, consider the atom $f^\checkmark$ within a behavioural type that binds f to $\texttt{foo}\,\alpha(\beta, \gamma)$. Assume to evaluate this type with $\Delta = \{\beta, \gamma\}$. That is, the two arguments are actually identical. What are the values of β and γ for evaluating $\texttt{foo}^\Delta_{\text{peak}}$ and $\texttt{foo}^\Delta_{\text{net}}$? 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 $\Psi(\beta)$ and $\Psi(\gamma)$ (but greater than any other value that is smaller);
3. to substitute β and γ with the result of 2.

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

The translation function. The translation function, called `translate`, is structured in three parts that respectively correspond to simple atoms, full behavioural 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. For simplicity we also let $\Delta(x) = x$ for every x that belongs to the local variables. 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 (unsynchronised) and about the state of vm names;
3. α is the name of the virtual machine of the current behavioural type;
4. $\overline{e}$ is the sequence of (over-approximated) costs of the current execution branch;
5. the behavioural type being translated; it may be either $\mathbb{a}$, $\mathbb{c}$ or $\overline{\mathbb{C}}$.

In the definition of `translate` we use the two functions

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

The left-hand side 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. The right-hand side 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 ($\alpha = \top$).

Finally, we will assume the presence of a lookup function `lookup` that takes method invocations $\texttt{m}\ \alpha(\overline{\mathbb{r}}, \overline{\beta})$ and returns tuples $\mathbb{c} : \mathfrak{o}, \texttt{R}$. This function is left unspecified.

The definition of `translate` follows. We begin with the translation of atoms.

$$\texttt{translate}[\Delta,\Psi,\alpha](\overline{e};e)(\mathbb{a}) \;=\; \begin{cases} (\Psi,\overline{e};e) & \text{when } \mathbb{a}=\texttt{0} \\ (\Psi[\beta\mapsto\varnothing\top],\overline{e};e;e+\texttt{CNEW}(\mathbb{t})) & \text{when } \mathbb{a}=\nu\beta \quad\text{and}\quad \Psi(\alpha)=\texttt{F}\mathbb{t} \\ (\Psi[\Delta(\beta)\mapsto\varnothing\bot],\overline{e};e;e+\texttt{CREL}(\mathbb{t})) & \text{when } \mathbb{a}=\beta^{\checkmark} \quad\text{and}\quad \Psi(\Delta(\beta))=\texttt{F}\mathbb{t} \\ (\Psi'[f\mapsto(\Psi|_{\Delta(\beta,\overline{\beta})},\texttt{R},\mathtt{m}\;\Delta(\beta)(\overline{\mathbb{r}},\Delta(\overline{\beta})\to\mathbb{o}))],\overline{e};e;e+f) & \text{when } \mathbb{a}=\nu f:\mathtt{m}\;\beta(\overline{\mathbb{r}},\overline{\beta})\to\mathbb{o} \\ & \text{and } \Psi'=\Psi[\beta\mapsto(\texttt{F}\cup\{f\})\mathbb{t}]^{\beta\in\texttt{R},\Psi(\beta)=\texttt{F}\mathbb{t}} \\ & \text{and } \texttt{lookup}(\mathtt{m}\;\Delta(\beta)(\overline{s\overline{e}},\Delta(\overline{\beta})))=\mathbb{c}:\mathbb{o},\texttt{R} \\ (\Psi\setminus f,\;(\overline{e};e)\sigma;(e)\sigma'+\textstyle\sum_{\gamma\in\Delta(\texttt{R}),\Theta(\gamma)=\texttt{F}\mathbb{t},\texttt{F}\neq\varnothing}\texttt{CREL}(\mathbb{t}) & \text{when } \mathbb{a}=f^{\checkmark} \quad\text{and}\quad \Psi(f)=(\Psi',\texttt{R},\mathtt{m}\;\beta(\overline{\mathbb{r}},\overline{\beta})\to\mathbb{o}) \\ & \text{and } \texttt{EqRel}(\beta,\overline{\beta})=\Xi \quad\text{and}\quad \Theta=\Psi'\searrow^{f}_{\downarrow}\Psi \\ & \text{and } \sigma=[\mathtt{m}^{\Xi}_{\text{peak}}(\overline{\mathbb{r}},\Theta(\Xi(\beta,\overline{\beta}))\Downarrow)/_{f}] \\ & \text{and } \sigma'=[\mathtt{m}^{\Xi}_{\text{net}}(\overline{\mathbb{r}},\Theta(\Xi(\beta,\overline{\beta}))\Downarrow)/_{f}] \\ & \text{and } \Psi''=\Psi[\gamma\mapsto\varnothing\bot]^{\gamma\in\texttt{R}}[\gamma'\mapsto\Theta(\beta)]^{\gamma'\in fv(\mathbb{o})\setminus\texttt{R}} \end{cases}$$

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 $f^{\checkmark}$. In this last case, let $\overline{e};e$ 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 is the purpose of the addend $\sum_{\gamma\in\Delta(\texttt{R}),\Theta(\gamma)=\texttt{F}\mathbb{t},\texttt{F}\neq\varnothing}\texttt{CREL}(\mathbb{t})$ that remove machines that are going to be removed by parallel methods (the constraint $\texttt{F}\neq\varnothing$) 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 $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta,\overline{\beta})$.

The of behavioural types is given by composing the definitions of the atoms. In this case, the output of `translate` is a set of cost equations.

$$\texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}) \;=\; \begin{cases} \{(se')\{\,\overline{e'}\,\} & \text{when } \mathbb{c}=\mathbb{a}\triangleright\varnothing \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',(se')\{\,\overline{e'}\,\}) \\ C'' & \text{when } \mathbb{c}=\mathbb{a}\,\S\,\mathbb{c}' \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',\{(se')\{\,\overline{e'}\,\}\}) \\ & \text{and } \mathrm{dom}(\Psi')\setminus\mathrm{dom}(\Psi)=\texttt{S} \\ & \text{and } \texttt{translate}(\Delta\cup\{\texttt{S}\},\Psi',\alpha,(se')\{\,\overline{e'}\,\},\mathbb{c}')=(\Psi'',C'') \\ C'\cup C'' & \text{when } \mathbb{c}=\mathbb{c}_1+\mathbb{c}_2 \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_1)=(\Psi',C') \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_2)=(\Psi'',C'') \\ C' & \text{when } \mathbb{c}=(se')\{\mathbb{c}'\} \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se\wedge se')\{\,\overline{e}\,\},\mathbb{c}')=(\Psi',C') \\ C' & \text{when } \mathbb{c}=(e')\{\mathbb{c}'\} \text{ and } e' \text{ contains } \_ \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}')=(\Psi',C') \end{cases}$$

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

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

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

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

$$[\Delta]=\{\alpha\mapsto\varnothing\alpha\mid\alpha=\Delta(\alpha)\}$$

$$\texttt{translate}(\Delta,[\Delta],\alpha_1)(0)(\mathbb{c})=\textstyle\bigcup_{i=1}^{n}(se_i)\{\,e_{1,i};\ldots;e_{h_i,i}\,\}$$

Then

$$\texttt{translate}(\Xi,\mathtt{m}\;\alpha_1(\overline{x},\alpha_2,\ldots,\alpha_k)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\texttt{R}) \;=\; \begin{cases} \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=0 & [\alpha_1=\bot] \\ \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{1,1} & [se_1\wedge\alpha_1\neq\bot] \\ \quad\vdots & \\ \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_1,1} & [se_1\wedge\alpha_1\neq\bot] \\ \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{1,2} & [se_2\wedge\alpha_1\neq\bot] \\ \quad\vdots & \\ \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_n,n} & [se_n\wedge\alpha_1\neq\bot] \\ \mathtt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=0 & [\alpha_1=\bot] \\ \mathtt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\ldots,\alpha_n]) & [\alpha_1=\partial] \\ \mathtt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_1,1} & [se_1\wedge\alpha_1=\top] \\ \quad\vdots & \\ \mathtt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_n,n} & [se_n\wedge\alpha_1=\top] \end{cases}$$

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

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

As an example, we show the output of `translate` when applied to the behavioural type of `double_release` computed in Section 4. Since `double_release` has two arguments, we generate two sets of equations, as discussed above. 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.

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

$$\texttt{translate}([\alpha_1,\alpha_2], \mathit{double\_release}\ \alpha_1(\alpha_2)\ \{\,\mathbb{c}\,\} : \_,\ \{\alpha_2\}) \ =$$

$$\begin{cases}
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 \neq \bot] \\
\\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = \mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) & [\alpha_1 = \partial] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 = \top]
\end{cases}$$

To highlight a cost computation concerning `double_release`, consider the following two potential users

```
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 ; }
```

which have corresponding behavioural types

$$\texttt{user1}\ \alpha(\,)\{\ \ \nu\beta\ \S\ \nu\gamma\ \S\ \ \nu\, f : \texttt{double\_release}\,\alpha(\beta,\gamma)\ \S\ f^{\checkmark}\ \}\ \texttt{-}\ ,\ \{\ \}$$

$$\texttt{user2}\ \alpha(\,)\{\ \ \nu\beta\ \S\ \ \nu\, f : \texttt{double\_release}\,\alpha(\beta,\beta)\ \S\ f^{\checkmark}\ \}\ \texttt{-}\ ,\ \{\ \}$$

The translations of the foregoing types give the set of equations

$$\texttt{translate}([\alpha_1], \mathit{user1}\ \alpha_1()\ \{\,\mathbb{c}_{\mathit{user1}}\,\} : \_,\ \{\}) \ =$$

$$\begin{cases}
\mathit{user1}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad + \mathit{double_release}_{\text{peak}}(\alpha_1,\top,\top) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad + \mathit{double_release}_{\text{net}}(\alpha_1,\top,\top) & [\alpha_1 \neq \bot] \\
\\
\mathit{user1}_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\text{net}}(\alpha_1) = \mathit{user1}_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user1}_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad + \mathit{double_release}_{\text{net}}(\alpha_1,\top,\top) & [\alpha_1 = \top]
\end{cases}$$

$$\texttt{translate}([\alpha_1], \mathit{user2}\ \alpha_1()\ \{\,\mathbb{c}_{\mathit{user2}}\,\} : \_,\ \{\}) \ =$$

$$\begin{cases}
\mathit{user2}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double_release}_{\text{peak}}(\alpha_1,\top) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double_release}_{\text{net}}(\alpha_1,\top) & [\alpha_1 \neq \bot] \\
\\
\mathit{user2}_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\text{net}}(\alpha_1) = \mathit{user2}_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user2}_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double_release}_{\text{net}}(\alpha_1,\top) & [\alpha_1 = \top]
\end{cases}$$

If we compute the cost of $\mathit{user1}_{\text{peak}}(\alpha)$ and $\mathit{user1}_{\text{net}}(\alpha)$ we obtain `2` and `0`, respectively. That is, in this case, `double_release` being invoked with two different arguments has cost `-2`. On the contrary, the cost of $\mathit{user2}_{\text{peak}}(\alpha)$ and $\mathit{user2}_{\text{net}}(\alpha)$ is `1` and `0`, respectively. That is, in this case, `double_release` being invoked with two equal arguments has cost `-1`.

6. Outline of the proof of correctness

The proof of correctness of our technique is long even if almost standard (see [12] 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 $cn \rightarrow cn'$ then cn' is well-typed as well. It is worth to observe that we cannot hope to demonstrate a statement guaranteeing type-preservation because our types are "behavioural" and change during the evolution of the systems. However, it is critical for the correctness of the cost analysis that there exists 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* $cn \rightarrow cn'$ *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. The basic remark here is that the number of alive virtual machines in a configuration is identical to the direct cost of the corresponding a behavioural type. This requires

3. the extension of the function `translate` to compute the cost equations for behavioural types of configurations. These equations allow us to 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 cost equations in [11]. This lets us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [11] or `PUBS` [1]. Given the basic cost and reduction cost inclusions, we can demonstrate the correctness theorem for our technique.

Theorem 6.4 (Correctness)**.** *Let* $\overline{M}\ \{\overline{F\,z}\ ;\ s'\}$ *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 $cn \rightarrow cn'$. 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. Related Work

After the pioneering work by Wegbreit in 1975 [21] that discussed a method for deriving upper-bounds costs of functional programs, a number of cost analysis techniques have been developed. Those ones that are closely related to this contribution are based either on cost equations (solvers) or on amortized analysis.

The techniques based on cost equations address cost analysis in three steps by: (*i*) extracting relevant information out of the original programs by abstracting data structures to their size and assigning a cost to every program expression, (*ii*) converting the abstract program into cost equations, and (*iii*) solving the cost equations with an automatic tool. Recent advances have been done for improving the accuracy of upper-bounds for cost equations [1, 4, 9, 11, 13] and we refer to [11] for a comparison of these tools. The main advantage of these techniques is that cost equations may carry Presburger arithmetic conditions thus supporting a precise cost analysis of conditional statements. The main drawback is that they extract control flow graphs from programs to perform their analysis, using abstract interpretations and control flow refinement techniques [4, 5]. It turns out that the above techniques do not provide the alias analysis and the name identity management that we have done in Section 4, which are essential for function or procedure abstraction, thus jeopardising compositional reasoning when large programs are considered.

The techniques based on amortized analysis [20] associate so-called potentials to program expressions by means of type systems (these potential determine the resources needed for each expression to be evaluated). The connection between the original program and the cost equations can be indeed demonstrated by a standard subject-reduction theorem [10, 15–18]. While the techniques based on types are intrinsically compositional and, more importantly, type derivations can be seen as certificates of abstract descriptions of functions, type based methods do not model the interaction of integer arithmetic with resource usage, thus being less accurate in some cases. An emblematic example is the following function:

```
Int foo(Int n, Int m, Vm x) {
  Fut<Int> f ;
  if (n==0) return 0;
  else if (n>m) { Vm v = new Vm ;
                  f = x!foo(n-1,m,v) ; f.get ; return 0 ;
  } else { Vm v = new Vm ; Vm w = new Vm ;
           f = v!foo(n-1,m,w) ; f.get ; return 0 ; }
}

{
  Vm x = new Vm ; Fut<Int> f = this!foo(2*n, n, x) ;
  f.get ;
}
```

which recursively invokes itself `2*n` times, and half the times executes the second branch – the case `(n>m)` – with cost `1` and half the times executes the else branch (with cost `2`). The techniques based on amortized analysis give a cost for the function `foo` that is `2*(2*n)=4*n` – without recognizing that the most costly branch is executed only half of the times – because they always assign the same cost to every branch. On the contrary, a more accurate analysis should derive that the actual cost of `foo` is `2*n + n = 3*n`. In fact, this is the case of solvers based cost equations, such as [11].

The technique proposed in this paper combines the advantages of the two approaches discussed above. It is modular, like the techniques based on cost equations, our one also consists of three steps, and it extracts the relevant information of programs by means of a *behavioural* type system, like the technique based on amortized analysis. Therefore, our technique is compositional and can be proved sound by means of a standard subject-reduction theorem. At the same time it is accurate in modelling the interaction of integer arithmetic with resource usage.

A common feature of cost analysis techniques in the literature is that they analyze *cumulative resources*. That is, resources that *do not decrease* during the execution of the programs, such as execution time, number of operations, memory (without an explicit `free` operation). As already discussed, this assumption eases the analysis because it permits to compute over-approximated cost. On the contrary, the presence of an *explicit or implicit* release operation entangles the analysis. In [2], a memory cost analysis is proposed for languages with garbage collection. It is worth to say that the setting of [2] is not difficult because, by definition of garbage collection, released memory is 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* that we have also used in Section 5. It is worth to notice that, for cumulative analysis, this two notions coincide while, 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 [6] has analysed the cost of a language with explicit releases. We observe that the release operation studied in [6] is used in a very restrictive way: only locally created resources can be released. This constraint guarantees that costs of functions are always not negative, thus permitting the (re)use of non-negative cost models of cumulative analysis.

We conclude by discussing cost analysis techniques for concurrent systems, which are indeed very few [3, 5, 14]. In order to reduce the imprecision of the analysis caused by the nondeterminism, [3, 5] use a clever technique for isolating sequential code from parallel code, called *may-happen-in-parallel* [8]. We notice that no one of these contributions consider a concurrent language with a powerful `release` operation that allows one remove the resources taken in input. In fact, 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.

8. 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 information about resource usages in programs, called behavioural types; an automatic translation that transforms these types into cost expressions; the application of solvers, like `CoFloCo` [11], 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 at least two lines of work. First, 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. Clearly, the management of these domains becomes more complex and the trade-off between simplicity and expressiveness must be carefully evaluated. Second, we intend to implement our analysis targeting a

programming language with a formal model as ABS [19], of which vml is a very basic sub-calculus. The current prototype translates a behavioural type program into cost equations and we expect to define an inference system that returns behavioural types in the same style of [12].

Acknowledgments

We thank Elena Giachino for the valuable discussions and for the help in the assessment of our technique with respect to those based on amortized analysis. We also thank Antonio Flores-Montoya and Samir Genaim for the discussions about the cost analysers CoFloCo and PUBS.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. 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.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. *SIGPLAN Not.*, 45(8):121–130, 2010.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost analysis of concurrent OO programs. In *Proceedings of APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[5] E. Albert, J. Correas, and G. 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] E. Albert, J. Correas, and G. Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of TACAS 2015*, Lecture Notes in Computer Science. Springer, 2015. To appear.

[7] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *Proceedings of SAS 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2012.

[8] R. 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] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. 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] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *Proceedings of SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[11] A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of 12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[12] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Software and Systems Modeling*, 2015. ISSN 1619-1366. . URL http://dx.doi.org/10.1007/s10270-014-0444-y. To appear.

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

[14] J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs, 2015. URL http://cs.yale.edu/homes/hoffmann/papers/parallelcost2014.pdfm. [Online; accessed 11-February-2015].

[15] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012. . URL http://doi.acm.org/10.1145/2362389.2362393.

[16] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL 2003*, pages 185–197. ACM, 2003.

[17] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *Proceedings of ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[18] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proceedings of CSL 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[19] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[20] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

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