Static analysis of cloud elasticity

Static analysis of cloud elasticity

Abel Garcia[a], Cosimo Laneve[a], Michael Lienhardt[b]

[a] *Department of Computer Science and Engineering, University of Bologna – INRIA Focus*
[b] *Computer Science Department, University of Torino, Italy*

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 record relevant information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioural types and returns 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.

Keywords: Resource consumption analysis, cloud computing, behavioural type system, subject reduction, static analysis, concurrent programming

1. Introduction

The analysis of resource usage in a program is of great interest because an accurate assessment could reduce, for example, 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, (*i*) have pricing models that allow one to hire on demand virtual machine instances and paying them for the time they are in use, and (*ii*) 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 [28], 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 language 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, 10]) and to make the connection between programs and closed-form expressions precise (see for instance [12, 20]). 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, the techniques are used in a very constrained way [5]. On the other hand, cloud computing elasticity needs powerful acquire operations *as well as* release ones (see for instance the *Amazon Elastic Compute Cloud* [26] or the *Docker Fiware*). Let us consider the following

Email addresses: abel.garcia2@unibo.it (Abel Garcia), cosimo.laneve@unibo.it (Cosimo Laneve), michael.lienhardt@di.unito.it (Michael Lienhardt)

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 [4]. 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 one machine is active and the other is released, or 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 types is defined in Section 5 by translating them in a code that is adequate for an off-the-shelf solver – the `CoFloCo` solver [13]. 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 6 we examine our prototype implementation and we report the results of some of our experiments. In particular, our technique excels in analyzing conditional branches, while it is less precise than multivariate amortized analysis [19] in presence of multi-variate arguments. We address this topic in Sections 6 and 7.

Our technique targets a simple concurrent language with explicit operations of creation and release of *virtual machines*. The language is defined in Section 2 and we discuss simplifications that ease the development of our technique in Section 3. In Section 8 we deliver concluding remarks. The correctness proof of the type system and with respect to the cost equations is reported in Appendix A. In the language we use the metaphor of cloud computing and virtual machines to illustrate our approach. However it is worth to 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) as long as these operations are "simple". For example, our technique might cover functions such as `malloc` and `free` as long as the sizes of memories that are allocated/freed are expressed in presburger arithmetics [9].

This article is a revised and enhanced version of [15] that includes a new semantics of the language, the full proofs of all the results and a discussion about our prototype implementation that includes the type inference system and the experimental assessments we did.

2. The language vml

The language vml is a future-based concurrent object-oriented language[1] with explicit acquire and release operations of virtual machines. In future-based languages, function/method invocations are *executed asynchronously* to the caller and are bound to variables called *futures*. The caller synchronises with the callee by using the future variables when the return value is strictly needed. The syntax and the semantics of vml are defined in the following two subsections; the third subsection discusses a number of examples.

Syntax. In vml we distinguish between *simple types* T, which are either integers `Int` or virtual machines `VM`, and *future types* `Fut<`T`>`, which type asynchronous invocations. The future argument T is instantiated with the return type of the invoked function. We use F to range over simple and future types. The notation $\overline{T\ x}$ denotes any finite sequence of *name declaration* $T\ x$ separated by commas. Similarly we write $\overline{T\ x\ ;}$ for a finite sequence of name declaration separated by semicolons.

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 $\{\overline{F\ z\ ;}\ s'\}$. The syntax of statements s, expressions z and pure expressions e of vml is defined by the following grammar:

s	::=	$x = z \mid \mathtt{if}\ e\ \{s\}\ \mathtt{else}\ \{s\} \mid \mathtt{return}\ e \mid s\ ;\ s \mid \mathtt{release}\ e$	statements
z	::=	$e \mid e\mathtt{!m}(\overline{e}) \mid e\mathtt{.get} \mid \mathtt{new\ VM()}$	expressions
e	::=	$\mathtt{this} \mid se \mid nse$	pure expressions

A statement s may be either one of the standard operations of an imperative language plus the `release` e operation that disposes the virtual machine e.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method call $e!m(\overline{e})$ where e is the virtual machine that will execute the call, and $\overline{e}$ are the arguments of the call. This invocation 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).

A (*pure*) expression e may be the reserved identifier `this`, a virtual machines identifier or an integer expression. 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 [9] (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:

nse	::=	$k \mid x \mid nse \le nse \mid \neg nse \mid nse\ \mathtt{and}\ nse \mid nse\ \mathtt{or}\ nse \mid nse + nse \mid nse - nse \mid nse \times nse \mid nse/nse$	non-size expressions
se	::=	$ve \mid ve \le ve \mid \neg se \mid se\ \mathtt{and}\ se \mid se\ \mathtt{or}\ se$	size expressions
ve	::=	$k \mid x \mid ve + ve \mid k \times ve$	integer size expressions
k	::=	*integer constants* $\mid$ *err*	

We will use the term $nse = nse'$ as an abbreviation of "$(nse \le nse')$ `and` $(nse' \le nse)$", and similarly for se. We notice that (non-size and size) expressions also contain the value *err* – see below the definition of $[\![e]\!]_l$ for the semantics of arithmetics expressions that contain *err*. 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 never have a continuation.

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

cn	::=	$\epsilon \mid fut(f, v) \mid vm(o, a, p, q) \mid invoc(o, f, \mathtt{m}, \overline{v}) \mid cn\ cn$	configurations
p	::=	$\{l \mid \epsilon\} \mid \{l \mid s\}$	process
q	::=	$\epsilon \mid p \mid q\ q$	sets of processes
v	::=	*integer constants* $\mid o \mid f \mid \bot \mid$ *err*	run-time values
a	::=	$\top \mid \bot$	machine states
l	::=	$[\cdots, x \mapsto v, \cdots]$	maps

[1] See [17] for more details on a similar language

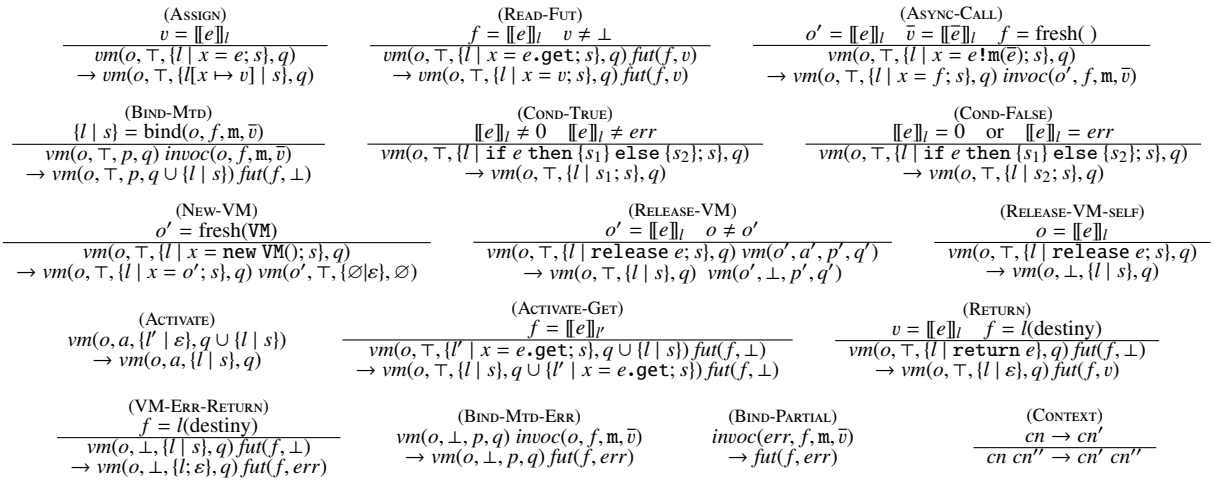

Figure 1: Semantics of `vml`.

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. The elements of configurations are

- *virtual machines* $vm(o, a, p, q)$ where o is a vm name; a is either $\top$ or $\bot$ depending on whether the machine is alive or dead; p is either $\{l \mid \epsilon\}$, representing a terminated statement, or $\{l \mid s\}$, representing an *active process*, where l maps each local variable to its value and s is the statement to execute; and q is the set of processes to evaluate.
- *future binders* $fut(f, v)$. When the value v is $\bot$ then the actual value of f has still to be computed.
- *method invocation messages* $invoc(o, f, \texttt{m}, \overline{v})$.

Runtime values v are either integers or virtual machines and future names, or $\bot$, meaning an un-computed value, or an erroneous value *err*. The following auxiliary functions are used in the semantic rules (we assume a fixed `vml` program):

- $\text{dom}(l)$ 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 names 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`. Arithmetic operations in `vml` are also defined on the value *err*: when one of the arguments is *err*, every arithmetic 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 name that is not defined in l.
- $\text{bind}(o, f, \texttt{m}, \overline{v}) = \{[\overline{x} \mapsto \overline{v}, \text{destiny} \mapsto f] \mid s[^o/_{\texttt{this}}]\}$, where $T\ \texttt{m}(\overline{T\ x})\{\overline{T'\ z}; s\}$ is a method of the program. We observe that, because of bind, the map l in processes $\{l \mid s\}$ also binds the special name destiny to a future value. We also observe that the local names $\overline{z}$ do not belong to $\text{dom}([\overline{x} \mapsto \overline{v}, \text{destiny} \mapsto f])$.

The transition relation rules are collected in Figure 1. The rules are almost standard, except those about the management of virtual machines and method invocations, which we are going to discuss.

Rule (New-VM) creates a virtual machine and makes it alive. Rules (Release-VM) and (Release-VM-self) dispose a virtual machine by means of the operation `release` x: this amounts to update its state a to $\bot$. Once a virtual machine has been released, no operation can be performed anymore by it, except letting the processes on the queue returning *err* – see rules (Activate), (VM-Err-Return), and (Bind-Mtd-Err).

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. Rule (Bind-Mtd) applies an invocation message by adding to the live callee virtual machine a process corresponding to the called method. In case the callee virtual machine is not live, either (Bind-Mtd-Err) or (Bind-Partial) is applied, which binds *err* to the future name. Rule (Read-Fut) allows the caller to retrieve the value returned by the callee. It is worth to notice that the semantics of `get` is different from that of `ABS` [25] or of [17] because it is not blocking with respect to other processes waiting to be executed on the same machine, see rule (Activate-Get). In fact, deadlock freedom is out of the scope of this contribution; in any case we refer to [17] for an algorithm (and a prototype) verifying deadlock freedom in `ABS`.

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

$$vm(start, \top, \{[\text{destiny} \mapsto f_{start}] \mid s[^{start}/_{\mathtt{this}}]\}, \varnothing)$$

where *start* is a special vm name and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$ and represents (part of) computations.

Examples. We illustrate `vml` by discussing few examples and, for every example, we also examine the 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!costly_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, the computation of `fact(n)` only requires 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 `z` and delivers the value `m*n`. Since every virtual machine creation occurs before any release operation, `costly_fact` will create as many virtual machines as the argument n. That is, if the available resources are k virtual machines, then `costly_fact` can compute factorials up-to k.

The analysis of `costly_fact` has been easy because the `release` operation is applied to a locally created virtual machine. Yet, in `vml`, `release` can be also applied to method arguments and the presence of this feature in concurrent codes is a 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 creates two virtual machines and releases the second one with `this!double_release(x,x)` before the recursive invocation.

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

The cost of `fake_method(n)` should be `n`. However this is not the case if `double_release` is under-approximated with cost `-2`: In such case, one would wrongly derive a cost `0` of `fake_method(n)`. In Section 6.2 we consider an erroneous `fake_method` that increases the argument of the recursive invocation (instead of decreasing it) and discuss the corresponding cost equations returned by our technique. We notice that, in this case, the amount of virtual machines used by the erroneous method is infinite. The aim of the following sections is to present a technique for determining the cost of method invocations. Such a technique has to be sensible to the identity, and to the state of method's arguments before and after the invocation.

Alive machines. Given a configuration cn, a term $vm(o, \top, p, q) \in cn$ is called *alive machine* in cn. Let $\texttt{alive}(cn)$ be the number of different alive machines in cn. The objective of this paper is to define a technique such that, given a configuration cn, it returns a n satisfying, for every $cn \longrightarrow^* cn'$, $\texttt{alive}(cn') \leq n$ (n is an upper bound to the alive machines in computations rooted at cn).

3. Determinacy of releases of method's arguments

Our cost analysis of virtual machines uses abstract descriptions that carry informations about concurrent method invocations and about creations and removals of virtual machines. In order to ease the compositional reasonings, method's abstract descriptions also define the arguments the method releases upon termination, called *method's effects*. In this contribution we stick to method descriptions that are as simple as possible, namely we assume that method's effects are *sets*. In turn, this requires methods' behaviours to be *deterministic* with respect to method's effects and, to enforce this determinacy, we define the following simplification properties.

The following simplifications allow us to focus on the relevant problems of the resource analysis of `vml`. It is possible to drop most of the simplifications by extending the approach presented in this article using almost standard solutions (for instance, with guarded or nondeterministic effects, see below). However these solutions would entangle a lot the technicalities of the paper, making more difficult its reading and comprehension.

Simplification 1: *the branches in a method body always release the same set of method's arguments.* For example, methods like

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

does not follow this simplification because the then-branch does not release anything while the else-branch releases the argument `x`.

This simplification is typical in type systems with (deterministic) effects. Usually people takes the least-upper bound of effects; however we cannot take least-upper bounds of removals because it would mean to under-estimate the machines effectively used. We cannot take the greatest lower bound of removals either, because it would mean to over-estimate the machines used and perform erroneous releases in the continuations. Indeed, to analyse the above program, one would just have to extend the approach presented in this article with *guarded effects*, i.e., mapping from execution branches to effects, which can precisely identify the effects of each execution branch of the method.

Simplification 2: *method invocations are always synchronized within caller's body.* This implies that method's effects occur upon method termination. For example, in

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

the method's effects of `foo2` is not deteministic because `double_release` might still be running *after* `foo2` termination. This means that the termination of `foo2` has no impact on the termination of `double_release`. Hence, the effect of `foo2` is empty – the caller cannot assume that `x` and `y` have been released upon synchronizing with `foo2`. Overall, this simplification supports a more precise analysis and can be easily dropped (at the cost of returning larger upper bounds).

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

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

```
Int foo4(VM x, VM y) {
     Fut<Int> f ; Int u ;
     f = x!simple_release(y) ;
     u = 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 and, in turn, it is not possible to determine whether `foo3` will release `x` or not. A similar non-determinism arises when the callee of a method releasing arguments is itself an argument. For example, in the above `foo4` method, it is not possible to determine whether `y` is released or not because we have no clue about `x`, being it an argument of `foo4`.

To analyse such programs, one would simply need to extend the approach presented in this article with non-deterministic effects.

Simplification 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 = y!identity(x) ; g = this!simple_release(y);
   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 non-determinism is caused by `identity`, which returns the argument that is going to be released by a parallel method.

To analyse programs with such methods, one would simply need to extend the approach presented in this article with non-deterministic effects.

Simplifications 1, 3, and 4 are enforced by the type system in Section 4, in particular simplification 1 by rule (T-Method), simplification 3 by rules (T-Invoke) and (T-Release), and simplification 4 by rules (T-Invoke) and (T-Return).

4. The behavioural type system of `vml`

Behavioural types are abstract codes highlighting the features of `vml` programs that are relevant for the resource 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* $\alpha, \beta, \gamma, \cdots$, 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 $\mathtt{F}, \mathtt{F}', \cdots$. We assume that *err* is a *special*

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

Figure 2: Behavioural Types Syntax

$$
\text{(T-Var)}\ \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad
\text{(T-Primitive)}\ \Gamma \vdash k : k \qquad
\text{(T-Op)}\ \frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1\ \mathtt{op}'\ e_2 : se_1\ \mathtt{op}'\ se_2}
$$

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

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

Figure 3: Typing rules for pure expressions

vm name representing the erroneous machine; therefore vm names will also range over *err*. The syntactic rules are presented in Figure 2.

Behavioural types $\nu\alpha$ and $\alpha^{\checkmark}$ express creations of virtual machines and their removal, respectively. The type $\nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}})$ defines method invocations and $f_{\mathbb{o}}{}^{\checkmark}$ defines the corresponding synchronisation with the computation of the future f, in this case $\mathbb{o}$ may be either α, which acts like a binder to the name of the returned virtual machine, or _ if the target of f returns a different type. The conditional type $(se)\{\mathbb{c}\}$ behaves like $\mathbb{c}$ whenever *se* is true; the type $\mathbb{c} + \mathbb{c}'$ models nondeterminism. We will always shorten the type $f_{\_}{}^{\checkmark}$ into $f^{\checkmark}$.

In order to have a more precise type of continuations, the leaves of behavioural types are labelled with *environments*, ranged over by $\Gamma, \Gamma', \cdots$. Environments are maps from method names $\mathtt{m}$ to terms $\alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R}$, from names 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. We assume that $\Gamma(err) = \varnothing\bot$. 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 $\mathtt{F}$ of future names 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 $\Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R}$ for methods.
- $\Gamma \vdash_{\mathtt{S}} z : \mathbb{x}, \mathbb{c} \triangleright \Gamma'$ for expressions z, where $\mathbb{x}$ is the value, $\mathbb{c}$ is the behavioural type for z and Γ' is the environment Γ *with updates* of names 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.

The index $\mathtt{S}$ in the judgments for expressions and statements defines the set of method's arguments – see rule (T-Method) – and is used in the rule (T-Return) in order to constrain that the returned machine, if any, does not belong to method's arguments (*cf.* Restriction 4 in Section 3).

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:

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

and, in Section 5, 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* $C[\ ]$ defined by the following syntax:

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

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

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. It is also worth to observe that, by rule (T-Primitive), $\Gamma \vdash err : err$ (while $\Gamma(err) = \varnothing\bot$). This expedient allows us to save one rule when typing method invocations with either erroneous or released carriers (*cf.* rule (T-Invoke-Bot)).

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-Return). Rule (T-Invoke) types method invocations $e!\mathtt{m}(\overline{e})$ by using a fresh future name. In particular, in the behavioural type $\nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}}) \to \beta$, the (fresh) future name f is associated to the the method, the vm name of the callee, the arguments *and to the returned value*. This last value is fundamental when the invocation returns a new machine because, in this case, the type acts as a binder of β. Another important remark is about 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 also 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. Let us now discuss the constraints in the second and third lines of the premise of (T-Invoke). As regards the second line, 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. 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 $e!\mathtt{m}(\overline{e})$.

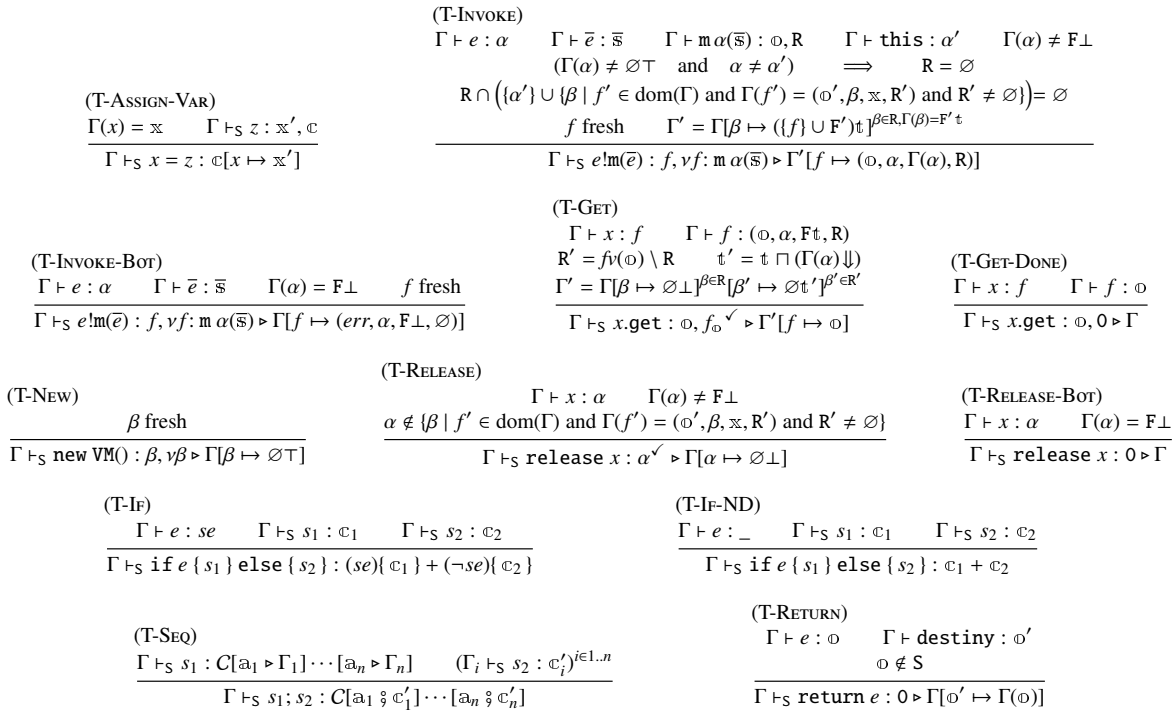

Figure 4: Type rules for expressions and statements.

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. There are two cases: either (*i*) $\mathtt{R} \neq \varnothing$ or (*ii*) $\mathtt{R} = \varnothing$. In case (*i*), by Restriction 3, $\mathtt{F}\mathbb{t}$, which is the value of the caller α when the method has been invoked, *is equal* to the value $\Gamma(\alpha)$ (and $\mathtt{F} = \varnothing$). In this case, if the method returns a new machine (*cf.* Restriction 4), its state must be $\varnothing\mathbb{t}$ (notice that $\mathbb{t} \sqcap \Gamma(\alpha) \Downarrow = \mathbb{t}$). The case (*ii*) is more problematic because the caller may be released by a method running in parallel and, therefore, the possible returned virtual machine must record this information in its state. Let $\mathtt{F}\mathbb{t}$ be the state of the caller (which is recorded in $\Gamma(f)$). We use the operation $\mathbb{t} \sqcap \Gamma(\alpha)\Downarrow$ to this purpose, which means that the returned machine gets the same state of the carrier α if no method is releasing α, otherwise its state is either $\varnothing\hat{o}$ or $\varnothing\alpha\downarrow$, according to the state of α was $\mathtt{F}\top$ or $\mathtt{F}\alpha$.

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.

Rule (T-Return) is a bit cryptic. First of all it applies provided $\mathtt{o} \notin \mathtt{S}$. In fact, by Restriction 4, the type of e in `return` e can be either _ or a virtual machine that has been created by the method. In both cases, these values do not belong to S, which is the set of virtual machines in method's arguments – see rule (T-Method). In particular, when the type of e is _, by (T-method), $\Gamma(\texttt{destiny}) = \_$ and, by definition, $\Gamma[\mathtt{o}' \mapsto \Gamma(\mathtt{o})] = \Gamma$.

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 $\mathtt{o}$ may be either _ or a fresh vm name ($\mathtt{o} \notin \{\alpha\} \cup \overline{\beta}$) 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 $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\mathtt{o})}$ guarantees that every branch of the behavioural 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

(T-Method)

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

(T-Program)

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

Figure 5: Behavioural typing rules of method and programs.

issues of our technique. The following listing shows the behavioural type of `double_release` and the step by step rule application for its calculation.

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

```
double_release α(β, γ) {  // instance of T-Method
    β✓ ⨟                   // instance of T-Release
    γ✓ ⨟                   // instance of T-Release and T-Seq
    0                      // instance of T-Return and T-Seq
} - , {β, γ}
```

The behavioural types of (`fact` and `costly_fact` examples from Section 2 are the following (we remove tailing `0` when irrelevant):

```
fact α(n) {
 (n==0){ 0 }
 +(n>0){ νx:fact α(n − 1) ⨟ x✓ }
} - , { }
```

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

we notice that the type of `costly_fact` records the order between the recursive invocation and the release of the machine. In the case of the behavioural type of `double_release` the key point is 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, we keep track of this with the "effects set" $\{\beta,\gamma\}$.

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 a solver of *cost equations*. These 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, and se is a size expression whose variables are contained in $\overline{x}$ (the syntax of exp and se is given in full detail in [13]).

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 $\mathtt{m}$, called *peak cost* of $\mathtt{m}$ and noted $\mathtt{m}_{\text{peak}}$, and the other returns the number of resources the method $\mathtt{m}$ creates without releasing, called *net cost* of $\mathtt{m}$ 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 peak cost of the sequential composition of two methods $\mathtt{m}$ and $\mathtt{m}'$ is the maximal value between $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$; while the cost of the parallel execution of $\mathtt{m}$ and $\mathtt{m}'$ is $\mathtt{m}_{\text{peak}} + \mathtt{m}'_{\text{peak}}$.

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

Arguments' identities. Consider the following code and the corresponding behavioural types

```
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;
}
```

$$\mathtt{simple\_release}\ \alpha(\beta)\{\ \beta^{\checkmark}\ \}\ \_\ ,\{\beta\}$$

$$\mathtt{m}\ \alpha(\beta,\gamma)\{\ \nu\, f : \mathtt{simple\_release}\ \alpha(\beta)\ ;\ \gamma^{\checkmark}\ ;\ f^{\checkmark}\ \}\ \_\ ,\{\beta,\gamma\}$$

We notice that, in the type of $\mathtt{m}$, there is not enough information to determine whether $\gamma^{\checkmark}$ will have a cost equal to `-1` or `0`. In fact, in the rule (T-Method), we assumed that the arguments were pairwise different. However, this is not the case for invocations. For instance, if $\mathtt{m}$ 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`. We solve this problem of arguments' identity in the analysis of behavioural types, in particular in the translation of method types. Namely, the above method $\mathtt{m}$ is translated in four cost functions: $\mathtt{m}^{\{1\},\{2\}}_{\text{peak}}(x,y)$ and $\mathtt{m}^{\{1\},\{2\}}_{\text{net}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\mathtt{m}^{\{1,2\}}_{\text{peak}}(x)$ and $\mathtt{m}^{\{1,2\}}_{\text{net}}(x)$, 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.) Then the translation of an invocation to a method $\mathtt{m}$ redirects the invocation to $\mathtt{m}^{\Xi}$, where Ξ is the equivalence relation expressing the identity of the arguments.

The function computing the equivalence relation of the arguments of an invocation is `EqRel`. `EqRel` takes a tuple of vm names and returns a partition of the indices of the tuple:

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

That is, if two indices are in the same set then the corresponding elements in the tuple are equal. So, for instance, $\mathtt{EqRel}(\alpha,\beta,\alpha) = \{\{0,2\},\{1\}\}$. We notice that, the possible outputs of $\mathtt{EqRel}(\alpha_0,\alpha_1,\alpha_2)$, without knowing the identities of α_0, α_1, and α_2, are

$\{\{0\}\{1\},\{2\}\}$	the three arguments are pairwise different
$\{\{0,1\},\{2\}\}$	the first and second argument are equal, the third is different
$\{\{0,2\},\{1\}\}$	the first and third argument are equal, the second is different
$\{\{0\},\{1,2\}\}$	the second and third argument are equal, the first is different
$\{\{0,1,2\}\}$	all the arguments are equal

Henceforth, 10 cost functions will correspond to a method with three arguments, 5 for its peak cost and 5 for its net cost.

Next, we also notice that the actual arguments of a $\mathtt{m}^{\{\{0,1\},\{2\}\}}$ are not three but two: in this case the second argument is useless. Therefore we need a notation for selecting the two relevant arguments in these cases. We use the notation (perhaps awkward) $\mathtt{EqRel}(\alpha,\beta,\alpha,\gamma)(\alpha,\beta,\alpha,\gamma)$ that actually returns the triple (α,β,γ).

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 Simplification 3 in Section 3, such an object cannot be released. This is the reason why, in the foregoing discussion about the method $\mathtt{m}$, we have 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 the rules of Figure 4, in order to enforce the restrictions in Section 3, we have already computed the state of every machine. In this section we recompute them 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 information that is needed at a later stage 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 $(\Psi', \mathtt{R}, \mathtt{m}\,\beta(\overline{se}, \overline{\beta}) \to \circledcirc)$. The translation environment Ψ' in $(\Psi', \mathtt{R}, \mathtt{m}\,\beta(\overline{se}, \overline{\beta}) \to \circledcirc)$ is only defined on vm names. For this reason, we call it *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:

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

This operation $\Psi \searrow^f_{\downarrow} \Psi'$ updates the vm-translation environment Ψ that is stored in the future f with the translation environment at the synchronisation point. It is worth to observe that, by the definition of our type system and the following translation function, the values of $\Psi(\alpha)$ and $\Psi'(\alpha)$ are related. In particular, if $\mathtt{t} = \alpha$ then $\mathtt{t}'$ 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 $\mathtt{t} = \top$ then $\mathtt{t}'$ 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

$$\mathtt{t} \otimes^\alpha \mathtt{t}' \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathtt{t} = \bot \text{ or } \mathtt{t}' = \bot \\ \alpha & \text{if } \mathtt{t} \text{ and } \mathtt{t}' \text{ are variables} \\ \alpha\downarrow & \text{otherwise} \end{cases} \qquad \mathtt{F}_1\mathtt{t}_1 \otimes^\alpha \mathtt{F}_2\mathtt{t}_2 \stackrel{def}{=} (\mathtt{F}_1 \cup \mathtt{F}_2)(\mathtt{t}_1 \otimes^\alpha \mathtt{t}_2)$$

Then

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

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

The operator $\otimes^\alpha$ has not been defined on vm values as ∂ or $\top$ because we merge vm names whose image by Ψ are either $\mathtt{F}\beta$ or $\mathtt{F}\beta\downarrow$ or $\mathtt{F}\bot$. As a notational remark, we observe that $\Psi(\Delta)$ is noted as a map $[\alpha_1 \mapsto \mathtt{F}_1\mathtt{t}_1, \cdots, \alpha_n \mapsto \mathtt{F}_n\mathtt{t}_n]$ instead of the standard notation $[^{\mathtt{F}_1\mathtt{t}_1, \cdots, \mathtt{F}_n\mathtt{t}_n}/_{\alpha_1, \cdots, \alpha_n}]$. 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 $\mathtt{foo}\,\alpha(\beta, \gamma)$. Assume to evaluate this type with $\Delta = \{\{\beta, \gamma\}\}$. That is, the two arguments are actually identical. Which are the values of β and γ for evaluating $\mathtt{foo}^\Delta_{\mathrm{peak}}$ and $\mathtt{foo}^\Delta_{\mathrm{net}}$? 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 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, behavioural types, and method types and full programs. This function 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 ($\alpha = \top$).

We will assume the presence of a lookup function `lookup` that takes method invocations $\mathtt{m}\ \alpha(\overline{\mathtt{r}}, \overline{\beta})$ and returns tuples $\mathbb{c} : \mathbb{o}, \mathtt{R}$. This function is left unspecified. We also write $\mathtt{R}[{}^{\mathbb{o}'}/_{\mathbb{o}}]$ to denote a set that is equal to $\mathtt{R}$ if $\mathbb{o} \notin \mathtt{R}$, that is equal to $(\mathtt{R} \setminus \{\mathbb{o}\}) \cup \{\mathbb{o}'\}$, otherwise.

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} = \mathtt{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) = \mathtt{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)) = \mathtt{F}\mathbb{t} \\ (\Psi'[f \mapsto (\Psi|_{\Delta(\beta,\overline{\beta})}, \mathtt{R}, \mathtt{m}\ \Delta(\beta)(\overline{\mathtt{r}}, \Delta(\overline{\beta}) \to \mathbb{o}))], \overline{e}; e; e + f) & \text{when } \mathbb{a} = \nu f : \mathtt{m}\,\beta(\overline{\mathtt{r}}, \overline{\beta}) \\ & \text{and } \Psi' = \Psi[\beta \mapsto (\mathtt{F} \cup \{f\})\mathbb{t}]^{\beta \in \mathtt{R}, \Psi(\beta) = \mathtt{F}\mathbb{t}} \\ & \text{and } \texttt{lookup}(\mathtt{m}\ \Delta(\beta)(\overline{s\mathtt{e}}, \Delta(\overline{\beta}))) = \mathbb{c} : \mathbb{o}, \mathtt{R} \\ (\Psi'' \setminus f,\ (\overline{e}; e)\sigma; (e)\sigma' + \sum_{\gamma \in \Delta(\mathtt{R}), \Theta(\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \texttt{CREL}(\mathbb{t})) & \text{when } \mathbb{a} = f_{\mathbb{o}'}{}^{\checkmark} \text{ and } \Psi(f) = (\Psi', \mathtt{R}, \mathtt{m}\,\beta(\overline{\mathtt{r}}, \overline{\beta}) \to \mathbb{o}) \\ & \text{and } \Theta = \Psi' \setminus^{f} \Psi \text{ and } \Theta(\texttt{EqRel}(\beta, \overline{\beta})) = (\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n) \\ & \text{and } \sigma = [\mathtt{m}^{\Xi}_{\text{peak}}(\overline{\mathtt{r}}, \mathtt{F}_1\mathbb{t}_1 \Downarrow, \cdots, \mathtt{F}_n\mathbb{t}_n \Downarrow) \Downarrow)/_f] \\ & \text{and } \sigma' = [\mathtt{m}^{\Xi}_{\text{net}}(\overline{\mathtt{r}}, \mathtt{F}_1\mathbb{t}_1 \Downarrow, \cdots, \mathtt{F}_n\mathbb{t}_n \Downarrow) \Downarrow)/_f] \\ & \text{and } \mathtt{R}' = \mathtt{R}[{}^{\mathbb{o}'}/_{\mathbb{o}}] \\ & \text{and } \Psi'' = \Psi[\gamma \mapsto \varnothing\bot]^{\gamma \in \mathtt{R}'}[\gamma' \mapsto \Theta(\beta)]^{\gamma' \in fv(\mathbb{o}') \setminus \mathtt{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 resources released by the method. This is the purpose of the addend $\sum_{\gamma \in \Delta(\mathtt{R}), \Theta(\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \texttt{CREL}(\mathbb{t})$ that removes 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 $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta, \overline{\beta})$.

The translation 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.

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

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

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

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

$$\Delta = \{\,\{\,\alpha_{i_1},\ldots,\alpha_{i_m}\,\} \mid \{\,i_1,\ldots,i_m\,\} \in \Xi\,\} \qquad \text{and} \qquad [\Delta] = \{\alpha \mapsto \varnothing\alpha \mid \alpha = \Delta(\alpha)\}$$

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

Then

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

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 can 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 and the corresponding behavioural types

```
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 ;
}
```

```
user1 α( ){
  νβ ; νγ ; ν f : double_release α(β,γ) ; f✓
} - , { }

user2 α( ){
  νβ ; ν f : double_release α(β,β) ; f✓
} - , { }
```

The translations of the foregoing types give the following 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`.

Theorem 5.1 (Correctness). *Let $\overline{M}\ \{\overline{F\,z}\ ;\ s'\}$ be a well-typed program and let $\overline{\mathbb{C}}$, $\mathbb{c}$ be its behavioural type and cn be its initial configuration. Let also n be a solution of the function* $\texttt{translate}(\overline{\mathbb{C}}, \mathbb{c})$. *Then, for every $cn \longrightarrow^* cn'$,* $\texttt{alive}(cn') \leq n$.

The proof of correctness of our technique is long, even if almost standard (see [17] for a similar proof). It consists of two parts:

(Inf-Method)

$$\frac{\Gamma(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \circledcirc, \mathtt{R} \qquad \mathtt{S} = \{\alpha\} \cup \overline{\beta} \qquad \circledcirc \notin \mathtt{S} \\ \Gamma[\mathtt{this} \mapsto \alpha][\mathtt{destiny} \mapsto \circledcirc][\overline{x} \mapsto \overline{x}][\overline{z} \mapsto \overline{\beta}][\alpha \mapsto \varnothing\alpha][\overline{\beta} \mapsto \varnothing\overline{\beta}] \vdash_{\mathtt{S}} s : C[\mathtt{a}_1 \triangleright \Gamma_1] \cdots [\mathtt{a}_n \triangleright \Gamma_n] \\ \left(\Gamma_i(\gamma) = \Gamma_j(\gamma)\right)^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\circledcirc)} \qquad \mathtt{R}' = (\mathtt{S} \cup fv(\circledcirc)) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}}{\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{Int}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\} \ : \mathtt{m}\ \alpha(\overline{x},\overline{\beta})\ \{C[\mathtt{a}_1 \triangleright \varnothing] \cdots [\mathtt{a}_n \triangleright \varnothing]\} : \circledcirc,\ \mathtt{R}'; \Gamma[\mathtt{m} \mapsto \alpha(\overline{x},\overline{\beta}) : \circledcirc, \mathtt{R}']}$$

(Inf-Program)

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

Figure 6: Behavioural inference rules of method and programs.

1. the first part 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.
2. the second part uses a notion of *direct cost of a behavioural type* (of a configuration), which is the number of virtual machines occurring in the type plus an over-approximation of those that will be created, and demonstrate that the direct cost never increases during the evolution of a system.

The details are reported in Appendix A.

6. The SRA tool

The analysis technique presented in this article has been prototyped [14]. In this section we discuss a number of technical details about the type system and about the translation of the cost equations in Section 5 into an input that is adequate to an off-the-shelf solver – the CoFloCo solver [13]. We also deliver a preliminary assessment of our prototype by discussing a number of examples.

6.1. Type inference

The system defined in Section 4 is a type-checking process, that is it compels the user to write the behavioural types of methods. In order to relieve users from writing verbose annotations, it is possible to turn the type checking system into a type inference. In fact, in the case of vml, this adaptation is not difficult because the only problematic issue is the inference of method's effects R. This issue is easily solved by changing the rules (T-Method) and (T-Program) in Figure 5 into those of Figure 6. In particular, rule (Inf-Method) uses a new judgment $\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{T}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\} : \mathtt{m}\ \alpha(\overline{x},\overline{\beta})\ \{\mathbb{c}\} : \circledcirc,\ \mathtt{R}'; \Gamma'$, which also returns a new environment Γ' and where $\mathtt{R}'$ may be different from the method's effects stored in $\Gamma(\mathtt{m})$. The main difference between (T-Method) and (Inf-Method) is that the latter one lets Γ' record method's effects computed in the premise. The key constraint in Figure 6 is the left premise of (Inf-Program). This premise imposes method definitions to be typed with the *same* input and output environment. That is, the environment Γ is a *fixpoint* of the rule.

In order to compute the fixpoint Γ in the rule (Inf-Program), we notice that the differences between Γ and Γ' in $\Gamma \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma'$, if any, are in methods' effects. In addition, these method's effects are all *finite* because they are always a subset of method's arguments. Therefore, there is a standard technique for computing the *least fixpoint* environment:

1. verify that the transformation $\Gamma \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma'$ is monotone with respect to the subset ordering;
2. let Γ_0 be such that $\Gamma_0(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \circledcirc, \varnothing$ and let $\Gamma_i \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma_{i+1}$. Then there exists a least n such that $\Gamma_n \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma_n$.

It is easy to verify that 1 holds. Henceforth, the correctness of [14]. For example, let

```
m1 α(β,γ,δ){
      νf : m2 α(β) ; f✓ ; γ✓
} - , { }
```

```
m2 α(β){
      νf : m3 α(β) ; f✓
} - , { }
```

```
m3 α(β){
      β✓
} - , { }
```

Then, the reader can verify that the computation of the least fixpoint environment gives the following sequence of environments:

- step 0: $\Gamma_0 = [\mathtt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \varnothing,\ \mathtt{m2} \mapsto \alpha(\beta) : -, \varnothing,\ \mathtt{m3} \mapsto \alpha(\beta) : -, \varnothing]$;
- step 1: $\Gamma_1 = [\mathtt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\gamma\},\ \mathtt{m2} \mapsto \alpha(\beta) : -, \varnothing,\ \mathtt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$;
- step 2: $\Gamma_2 = [\mathtt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\gamma\},\ \mathtt{m2} \mapsto \alpha(\beta) : -, \{\beta\},\ \mathtt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$;
- step 3: $\Gamma_3 = [\mathtt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\beta,\gamma\},\ \mathtt{m2} \mapsto \alpha(\beta) : -, \{\beta\},\ \mathtt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$.

6.2. The translation of cost equations

To comply with `CoFloCo` input formats, we need to encode the vm values and the functions `CNEW` and `CREL` into integer values and cost equations, respectively. Let

– $\top$ be encoded by `1`, ∂ be encoded by `2`, and $\bot$ be encoded by `3`. The vm value $\alpha\downarrow$ is encoded by the conditional value $[\alpha = 3]3 + [1 \le \alpha \le 2]2$;

– `CNEW` and `CREL` equations are encoded as follows and are fixed for all programs:

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

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

In order to illustrate the encoding in `CoFloCo` of the cost equations in Section 5, we discuss the equations of the two factorial programs `fact` and `cheap_fact` in Section 2. The `CoFloCo` equations are reported in Figure 7.

```
1  eq(fact_peak(A,B), 0, [], [A = 3]).
2  eq(fact_peak(A,B), 0, [], [B = 0]).
3  eq(fact_peak(A,B), 0, [], [B > 0]).
4  eq(fact_peak(A,B), 0, [fact_peak(1,B)], [B > 0]).
5  eq(fact_peak(A,B), 0, [fact_net(1,B)], [B > 0]).
6
7  eq(fact_net(A,B), 0, [], [A = 3]).
8  eq(fact_net(A,B), 0, [fact_peak(A,B)], [A = 2]).
9  eq(fact_net(A,B), 0, [], [A = 1, B = 0]).
10 eq(fact_net(A,B), 0, [fact_net(1, B-1)], [A = 1, B > 0]).
11
12 eq(costly_fact_peak(A,B), 0, [], [A = 3]).
13 eq(costly_fact_peak(A,B), 0, [], [B = 0]).
14 eq(costly_fact_peak(A,B), 0, [], [B > 0]).
15 eq(costly_fact_peak(A,B), 0, [cnew(A)], [B > 0]).
16 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_peak(1, B-1)], [B > 0]).
17 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1)], [B > 0]).
18 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)], [B > 0]).
19
20 eq(costly_fact_net(A,B), 0, [], [A = 3]).
21 eq(costly_fact_net(A,B), 0, [costly_fact_peak(A,B)], [A = 2]).
22 eq(costly_fact_net(A,B), 0, [], [A = 1, B = 0]).
23 eq(costly_fact_net(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)], [A = 1, B > 0]).
```

Figure 7: Cost equations of `fact` and `costly_fact` in `CoFloCo` format

These equations do not require any comments because they exactly correspond to the output of the `translate` function in Section 5. We just observe that, in addition to these equations, one needs to specify a so-called *entry point*, which corresponds to the main body. This entry point has the format

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

that is instantiated into `entry(main(MAINVM):[])` for the main body. It is also possible to add ad-hoc entries for particular methods, if the user wants to highlight their cost (in particular, our translator adds entries for peak and net costs of every method). For example, the following table reports the outputs (of `CoFloCo`) corresponding to the entries in the left column.

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

The *net cost* of `fact` and `costly_fact` is equal to `0` because, in both cases, every created virtual machine is released before the end of the program. For *peak cost*, the number of virtual machines in `fact` is 0. On the other hand, `costly_fact` creates at each step a virtual machine and releases it after the recursive call. This management of virtual machines gives a *peak cost* equal to maximum between `1` and `B`.

Unbounded recursive method with release operations. Consider the following wrong version of the `fake_method` example from Section 2, where the programmer has erroneously written "$n + 1$" instead of "$n - 1$" in the recursive invocation (henceforth the recursion is unbounded because the base case is never met). As a result one can expect that the number of created virtual machines is infinite because, at every step, only one of the two created machines is released.

```
Int fake_method(Int n) {
       if (n=0) return 0 ;
       else { VM x, y ; Fut<Int> f, g; Int u, v;
              x = new VM() ; y = new VM() ;
              f = this!double_release(x,x) ; u = f.get ;
              g = this!fake_method(n+1) ; v = g.get ;
              return 0 ; }
}
```

The equations generated by the analysis are shown in Figure 8. The results of the analysis of these equations produce a cost of "infinity" for the `main` method, as suspected. However, lets suppose the situation in which both vms are released (change `double_release(x,x)` to `double_release(x,y)`). In this case the set of equations of `fake_method` remains almost identical with the only difference that the element (`crel(1)` in equations 10, 12, 14 and 18, now appears two times). If we re-run the cost analysis with the new equations we obtain, instead of infinity, a maximum cost of two vms for the `main` method. Again this is the expected behavior because even though the program runs indefinitely there are no more than two vms (in addition to the main one) co-existing at the same time.

There is an issue about Figure 7 that deserves to be commented. `CoFloCo` outputs an error when it is fed with that set of equations. This is because instructions 8 and 21 are mutually recursive with instructions 5 and 17-18, respectively, and mutual recursion is banned by the analyser. It is worth to notice that 8 and 21 correspond to invocations of `fact_net` and `costly_fact_net` when the carrier is ∂. Intuitively, the idea behind these equations is that when the vm state is unknown the analyzer should assume the worth and consider the highest possible cost (the peak cost) as the net cost. The workaround to solve this problem is to remove the mutual recursion by replacing *in-place* the internal function invocation by its body.

6.3. Assessments – a comparison with `C4B` *and* `SACO`

There are very few tools targeting resource analysis in programs that feature concurrency and negative costs due to explicit release operations. To the best of our knowledge, the most relevant tool in the literature is `SACO`[2], which is based on the results of [4] and is able to address a setting similar to ours. We also compare `SRA` with `C4B`[3] [11], a tool based on amortized analysis and using the results in [19]. Notwithstanding `C4B` only targets sequential programs, it may be considered as the state-of-the-art of *certified tools* for the automatic static analysis of resource usage bounds.

Some remarks on the tool comparison are in order:

[2] Web site at `http://ei.abs-models.org:8082/clients/web/`

[3] Web site at `http://www.cs.yale.edu/homes/qcar/aaa/`

```
eq(fakeMethod01peak(THISVM,N), 0, [], [THISVM = 3]).
eq(fakeMethod01peak(THISVM,N), 0, [], [THISVM < 3]).

eq(fakeMethod01net(THISVM,N), 0, [], [THISVM = 3]).
eq(fakeMethod01net(THISVM,N), 0, [], [N = 0, THISVM = 1]).

eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM)], [N > 0, THISVM < 3]).
eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM)], [N > 0, THISVM < 3]).
eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011peak(THISVM, 2)], [N > 0, THISVM < 3]).
eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
                                   crel(1)], [N > 0, THISVM < 3]).
eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
                                   crel(1),fakeMethod01peak(THISVM, N + 1)], [N > 0, THISVM < 3]).
eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
                                   crel(1),fakeMethod01net(THISVM, N + 1)], [N > 0, THISVM < 3]).

eq(fakeMethod01net(THISVM,N), 0, [], [THISVM = 3]).
eq(fakeMethod01net(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
                                   crel(1),fakeMethod01net(THISVM, N + 1)], [N > 0, THISVM = 1]).

eq(doubleRelease011peak(THISVM,X), 0, [], [THISVM = 3]).
eq(doubleRelease011peak(THISVM,X), 0, [], [THISVM < 3]).
eq(doubleRelease011peak(THISVM,X), 0, [crel(X)], [THISVM < 3]).
eq(doubleRelease011peak(THISVM,X), 0, [crel(X),crel(3)], [THISVM < 3]).

eq(doubleRelease011net(THISVM,X), 0, [], [THISVM = 3]).
eq(doubleRelease011net(THISVM,X), 0, [crel(X),crel(3)], [THISVM = 1]).
```

Figure 8: Cost equations of `fake_method` and `double_release` in `CoFloCo` format

– The three tools targets three different programs: `C4B` targets `C`, `SACO` targets `ABS`, and `SRA` targets `vml`. Therefore, in order to compare the three tools, every test program has been rewritten according to the corresponding language of the tool. The sources of these tests are available in github [4].

– Every tool uses different metrics for quantitative analysis. `SRA` quantifies the resource usage by means of the operations `new VM()` and `release`. `C4B` has two metrics: the (*i*) back-edge metrics that assigns a cost of 1 to every back edge in the control-flow graph, thus not allowing negative costs, and (*ii*) the tick metric that uses a special operation `tick(n)` that has a cost of n where n can be either a positive or negative number. `SACO`, on the other hand, considers both of these kinds of metrics and, in addition, it also support other ones that are specific to the domain of the targeted language.

– Both `SACO` and `SRA` separate *peak costs* and *net costs*. This distinction is not made in `C4B` because it does not address concurrency. Therefore we restrict the results of Table 9 only to the *net costs* of the corresponding programs.

Table 9 reports the results of our analysis. It is worth to observe that `SRA` returns better results than both `C4B` and `SACO` for the program `betterThanAmortized`. This program, reported in Figure 10, defines a method `betterThanAmortised(`n`,`m`,`x`)` that recursively invokes itself and, when ($n > m$) every invocation has cost 1, otherwise it has cost 2. Therefore, the cost of `betterThanAmortised(`$2*n$`,`n`,`x`)` is $3*n$ and this is the result of `SRA`. However, the other techniques do not recognize that the most costly branch is executed only half of the times and return $2*(2*n) = 4*n$. Such precision is achieved in `SRA` by means of the treatment given to numerical expressions, and by the use of a solver (as [13]) capable to deal with conditional branches accurately. This precision is lost in techniques based on type systems because of the unifications that are necessary to enforce on the branches of conditionals, in order to type the continuations. We also observe that `betterThanAmortized` is not an exotic, ad-hoc program but a rather common programming pattern (a sequence of nested conditional statements).

`SRA` also returns remarkable results (*i*) in presence of unbounded recursion – see the example `producerConsumer` –, (*ii*) in cases where the identity of the arguments is relevant for the cost analysis – see the example `doubleRelease` –, and (*iii*) for standard patterns of concurrency – see the examples `doWorkAsync` and `producerConsumer`.

[4] https://github.com/abelunibo/SRA-Tests-Files)

Program	C4B	SACO	SRA
betterThanAmortised	$2 * n$	$2 * n$	$n + m$
greatestCommonDivisor	$Max(n, m)$	*failed*	$n + m$
costlyFactorial	n	n	n
doWorkSync	2	2	2
quicksort	$2 * n$	*not analysable*	*not analysable*
SPEED1	$2 * n$	*not analysable*	*infinite*
doWorkASync	*not analysable*	n	n
producerConsumer	*not analysable*	*not analysable*	3
doubleRelease	*failed*	-2	$[x = y]$-1; $[x \neq y]$-2

Figure 9: Comparison of C4B, SACO, and SRA for some simple programs

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

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

Figure 10: The program betterThanAmortized

As a main downside of SRA, it is unsatisfactory for computing costs in presence of multi-variate arguments, as in greatestCommonDivisor and SPEED1. However, it is worth to observe that this loss of precision is actually a limitation of the cost equation solver that cannot handle multi-variate arguments in its current version. For a similar reason, SRA fails in computing the cost of the quicksort program because of a restriction in the CoFloCo solver that does not handle non-linear recursion (while SRA correctly derives the behavioural types of quicksort).

7. Related Work

After the pioneering work by Wegbreit in 1975 [28] 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 [2, 3, 10, 13, 18] and we refer to [13] 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. As a downside, techniques that extract control flow graphs and use control flow refinement techniques (such as [3, 4]) have a less precise alias analysis and name identity management than our technique (Section 4). These two features are

essential for function or procedure abstraction; in fact, a weak approach would jeopardise compositional reasoning when large programs are considered.

The techniques based on amortized analysis [27] associate so-called potentials to program expressions by means of type systems (these potentials 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 [12, 20, 22–24]. 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 (such as the program `betterThanAmortised` in Figure 10).

Our technique 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 [6], a memory cost analysis is proposed for languages with garbage collection. It is worth to say that the setting of [6] 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 [5] has analysed the cost of a language with explicit releases. We observe that the release operation studied in [5] 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 [1, 4, 21]. In order to reduce the imprecision of the analysis caused by the nondeterminism, [1, 4] 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 [4], 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` [13], 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, such as those based on multivariate amortized analysis [19].

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 [25], of which vml is a very basic sub-calculus. The current prototype translates a behavioural type program into cost equations.

Our behavioural types are abstract specifications of programs that highlight resource usages. In our mind, these abstract specifications must bridge the gap between programs and performance constraints in Service Level Agreement (SLA, in short) documents. As discussed in [16], vml (actually, full ABS) is intended to be a language using Amazon Elastic Cloud Computing features as library functions. Therefore, vml might be used to model cloud services, where a primary issue is the compatibility with the SLA document. The technique conveyed by the present article is to (*i*) extract the abstract specifications by means of the type (inference) system, (*ii*) compute the cost by means of the solver, and (*iii*) verify whether the computed cost complies with the SLA or not. This paper demonstrates that these steps are all correct. In this respect, a relevant future issue is the integration of our technique with other solvers, such as theorem provers, to be used in those cases where the cost equation solver is too imprecise.

Acknowledgements. 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] Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Román-Díez, G., 2015. Object-sensitive cost analysis for concurrent objects. Software Testing, Verification and Reliability 25 (3), 218–271, stvr.1569.
URL http://dx.doi.org/10.1002/stvr.1569

[2] Albert, E., Arenas, P., Genaim, S., Puebla, G., 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In: Proceedings SAS 2008. Vol. 5079 of Lecture Notes in Computer Science. Springer, pp. 221–237.

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

[4] Albert, E., Correas, J., Román-Díez, G., 2014. Peak cost analysis of distributed systems. In: Proceedings of SAS 2014. Vol. 8723 of Lecture Notes in Computer Science. Springer, pp. 18–33.

[5] Albert, E., Correas, J., Román-Díez, G., 2015. Non-Cumulative Resource Analysis. In: Proceedings of TACAS 2015. Lecture Notes in Computer Science. Springer, pp. 85–100.

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

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

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

[9] Berman, L., 1980. The complexity of logical theories. Theoretical Computer Science 11 (1), 71 – 77.
URL http://www.sciencedirect.com/science/article/pii/0304397580900377

[10] Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J., 2014. Alternating runtime and size complexity analysis of integer programs. In: Proceedings of TACAS 2014. Vol. 8413 of Lecture Notes in Computer Science. Springer, pp. 140–155.

[11] Carbonneaux, Q., Hoffmann, J., Shao, Z., 2015. Compositional certified resource bounds. In: Proceedings of the 36th PLDI Conference. ACM, pp. 467–478.

[12] Chin, W., Nguyen, H. H., Qin, S., Rinard, M., 2005. Memory usage verification for oo programs. In: Proceedings of SAS 2005. Vol. 3672 of Lecture Notes in Computer Science. Springer, pp. 70–86.

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

[14] Garcia, A., Laneve, C., 2015. Static analyzer of resource usage upper bounds.
URL sra.cs.unibo.it

[15] Garcia, A., Laneve, C., Lienhardt, M., 2015. Static analysis of cloud elasticity. In: Proceedings of the 17th PPDP. ACM, pp. 125–136.

[16] Giachino, E., de Gouw, S., Laneve, C., Nobakht, B., 2016. Statically and dynamically verifiable sla metrics. To appear on Lecture Notes in Computer Science.
URL cs.unibo.it/~laneve/papers/SLA-metrics.pdf

[17] Giachino, E., Laneve, C., Lienhardt, M., 2015. A framework for deadlock detection in ABS. Software and Systems ModelingTo appear.
URL http://dx.doi.org/10.1007/s10270-014-0444-y

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

[19] Hoffmann, J., Aehlig, K., Hofmann, M., 2011. Multivariate amortized resource analysis. In: Proceedings of POPL 2011. ACM, pp. 357–370.

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

[21] Hoffmann, J., Shao, Z., 2015. Automatic static cost analysis for parallel programs. [Online; accessed 11-February-2015]. URL http://cs.yale.edu/homes/hoffmann/papers/parallelcost2014.pdf
[22] Hofmann, M., Jost, S., 2003. Static prediction of heap space usage for first-order functional programs. In: Proceedings of POPL 2003. ACM, pp. 185–197.
[23] Hofmann, M., Jost, S., 2006. Type-based amortised heap-space analysis. In: Proceedings of ESOP 2006. Vol. 3924 of Lecture Notes in Computer Science. Springer, pp. 22–37.
[24] Hofmann, M., Rodriguez, D., 2009. Efficient type-checking for amortised heap-space analysis. In: Proceedings of CSL 2009. Vol. 5771 of Lecture Notes in Computer Science. Springer, pp. 317–331.
[25] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M., 2011. ABS: A core language for abstract behavioral specification. In: Proceedings of FMCO 2010. Vol. 6957 of Lecture Notes in Computer Science. Springer, pp. 142–164.
[26] Mathew, S., November 2014. Overview of amazon web services, at d0.awsstatic.com/whitepapers/aws-overview.pdf.
[27] Tarjan, R. E., 1985. Amortized computational complexity. SIAM Journal on Algebraic Discrete Methods 6 (2), 306–318.
[28] Wegbreit, B., 1975. Mechanical program analysis. Communications of the ACM 18 (9), 528–539.

Appendix A. The proof of correctness of the type system

Appendix A.1. Type system correctness

As discussed in Section 5, the correctness of our analysis technique consists of two parts. The first part addresses the correctness of the type system in Section 4, which is usually expressed by a subject reduction theorem. This theorem states 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 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 and the type of cn'.

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.

The extension of the typing to configurations. When typing configurations there are two kinds of vm names: *static-time vm names* and *runtime vm names*. The former names are ranged over by $\alpha, \beta, \cdots$; the latter names, which also include *err*, are ranged over by $o, o', \cdots$. With an abuse of notation, in the following, α and $\mathfrak{o}$ will also range over runtime vm names o and *err*. This abuse will let us to reuse the typing rules in Figure 4 (with one exception, see below). The syntax of runtime types is:

$\mathbb{k}$	::=	$\mathtt{0} \mid f(\bot) \mid f(\_) \mid f(o) \mid (f, \mathtt{m}\, o(\overline{\mathtt{r}})) \mid vm(o, a \mid \mathbb{f}; \overline{\mathbb{f}}) \mid \mathbb{k} \parallel \mathbb{k}$	runtime types
$\mathbb{f}$	::=	$(f, \mathbb{c}, \mathsf{S})$	process types
$\mathbb{a}$	::=	$\ldots \mid [\mathfrak{o}]$	extended atom

The syntactic category $\mathbb{k}$ models behavioural types only existing at runtime, in particular, virtual machines, futures, and processes. The term $\mathtt{0}$ correspond to an element of the running program that does not have any effect on its cost (*e.g.*, ϵ); $f(\bot)$, $f(\_)$, and $f(o)$ type a future binder: f is the name of the future, $\bot$ or _ or o identify the value of the future, namely either $\bot$, if it is still not computed, or _ or o, if it has been already computed (see the rest of the paragraph for more details); $(f, \mathtt{m}\, o(\overline{\mathtt{r}}))$ corresponds to an invocation message. The type $vm(o, a \mid \mathbb{f}; \overline{\mathbb{f}})$ corresponds to a virtual machine (alive if $a = \top$, dead otherwise) executing the process $\mathbb{f}$ and with a set of processes $\overline{\mathbb{f}}$ to execute. Processes are triples $(f, \mathbb{c}, \mathsf{S})$ where:

– f is the identity of the process (i.e. the name of its future);

– $\mathbb{c}$ is the type of the statement the process will execute;

– S is a set of runtime vm names.

The term $\mathbb{k} \parallel \mathbb{k}$ types the parallel composition of runtime configurations. Atoms are extended with the term $[\mathfrak{o}]$ that represents the returned value of a method – see the discussion below.

The runtime type system consists of rules in Figure 4 plus the rules in Figure A.11. The reader may notice that the first rule (BT-Return) replaces the definition of the judgment $\Gamma \vdash_{\mathsf{S}} s : \mathbb{c} \triangleright \Gamma'$ for the `return` statement in Figure 4. This modification, together with the extensions of atoms, is used to refine the (static-time) typing rules in order to deal

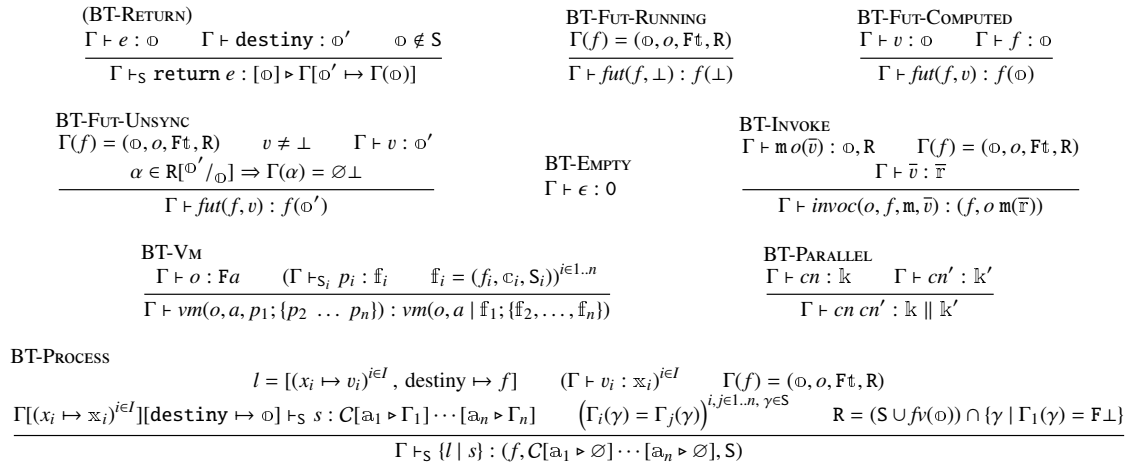

Figure A.11: Runtime Typing rules

with a naming issue in the subject reduction theorem. In particular, when a method returns a freshly created virtual machine, we associate to it a fresh name in (T-Invoke) that we can use later in (T-Get) and in the rest of the typing of the (static) program. The problem is that, when we type the method invocation (i.e. when we collect the information about the method and its return value in order to type future usage of this value), the identity of the returned virtual machine is still unknown because the virtual machine has not yet been created. In order to solve this problem, we extend atoms with the term $[\mathbb{o}]$, which retains the returned value and we use the new rule (BT-Return) to type the `return` statements.

The other rule of Figure A.11 that we comment is (BT-Vm). This rule allows one to type a vm term containing a number of processes p. To this aim, one has to type every p by means of the judgment $\Gamma[\texttt{this} \mapsto o] \vdash_{\mathtt{S}} p : (f, \mathbb{c}, \mathtt{S})$; namely the index $\mathtt{S}$ and the third argument of the process type must be the same. In fact, this argument records the actual vm names of the corresponding method invocation – see rule (R-Bind-Mtd) in Figure A.12. In turn, this set is used in (BT-Process) to type the statements of the processes.

Lemma Appendix A.1 (Substitution Lemma)**.** *Let $\Gamma \vdash cn : \mathbb{k}$ and let ι be an injective renaming of names, vm names, and future names. Let $\iota(\Gamma) \stackrel{def}{=} [\iota(x) \mapsto \iota(\Gamma(x))]^{x \in \mathrm{dom}(\Gamma)}$.*

Then $\iota(\Gamma) \vdash \iota(cn) : \iota(\mathbb{k})$. Similarly for $\Gamma \vdash_{\mathtt{S}} \{l \mid s\} : (f, \mathbb{c})$ and $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c} \triangleright \Gamma'$ (in these two cases also $\mathtt{S}$ is mapped by ι) and for $\Gamma \vdash e : \mathbb{x}$.

Proof. The proof proceeds straightforward by induction on the structure of s. □

Runtime Types Evolution. As usual with behavioural types, our types change while the programs execute. These changes can be defined and, more importantly, can be related to the cost analysis. The modifications of types, noted $\rightsquigarrow$, is defined as a reduction relation in Figure A.12. The rules are mostly an adaptation of the operational semantics of `vml` at the type level. Therefore, to ease the understanding of the relation $\rightsquigarrow$, we use similar names for similar rules. Nevertheless, there are few differences from the operational semantics of `vml`: *i*) rule (Assign) has no corresponding rule in the definition of $\rightsquigarrow$ (assignments are managed at the type system level); *ii*) rule (Activate) is replaced by the rule (R-GC) (there are no scheduling policies in our runtime types, so the only effect of the rule (Activate) at the type level is the deletion of a finished process); iii) similarly, rule (Activate-Get) has no corresponding rule in the definition of $\rightsquigarrow$; and iv) because at runtime all objects identities are known, the rules (Read-Fut), (Async-Call), (Bind-Mtd) and (New-VM) have been adjusted in order to replace the names generated a static time with the objects' actual identities.

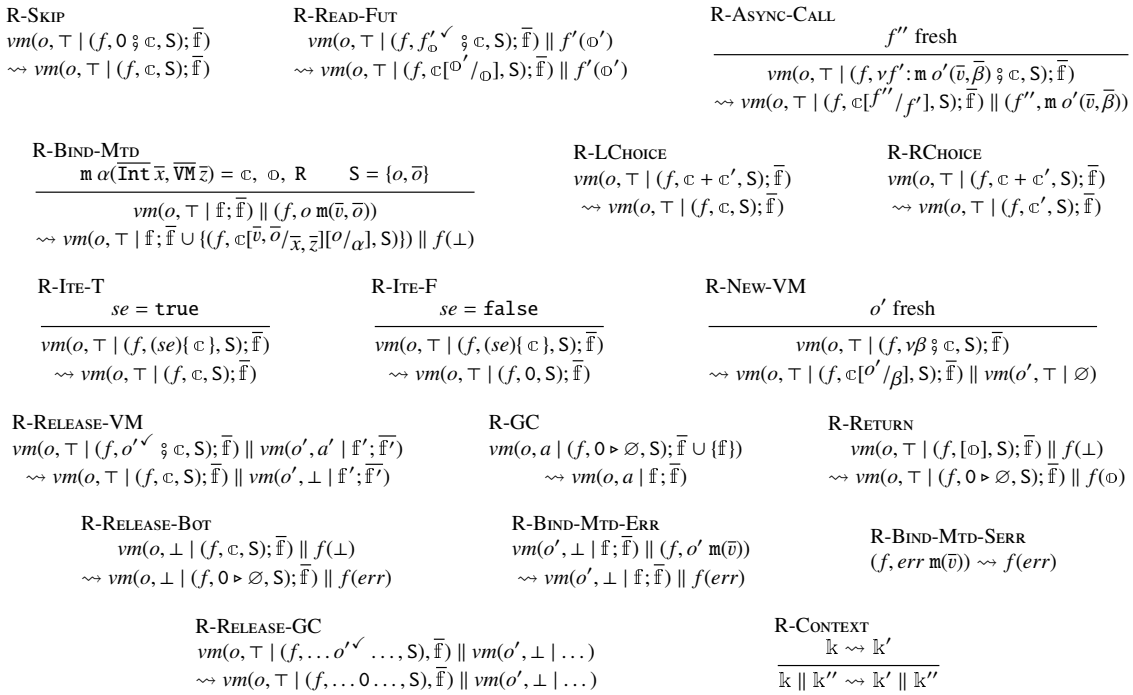

Figure A.12: Operational Semantics of Runtime Types

The subject reduction theorem. Based on the previous construction, we present the subject reduction theorem. Lemma Appendix A.2 relates the static time type system to the runtime type system: this corresponds to the first step of the subject reduction property.

Lemma Appendix A.2. *Let* $P = \overline{M}\ \{\overline{F\ x}\ ;\ s\}$ *be a* `vml` *program, and let* $\Gamma \vdash P : \overline{\mathbb{C}}, \mathbb{c}$. *Then*

$$\Gamma[start \mapsto \top][f_{start} \mapsto (\_, start, \varnothing\top, \varnothing)] \vdash vm(start, \top, \{[destiny \mapsto f_{start}] \mid s\}, \varnothing) : vm(start, \top \mid (f_{start}, \mathbb{c}, \varnothing); \varnothing)\ .$$

Proof. The proof follows by the application of rules (BT-Vm) and (BT-Process), and using the hypothesis. □

The main subject reduction theorem states that any well-typed runtime program reduces into another well-typed runtime program.

Theorem Appendix A.3 (Subject Reduction)**.** *Let* $\Gamma \vdash cn : \mathbb{k}$. *If* $cn \to cn'$ *then there exists* Γ' *such that* $\Gamma' \vdash cn' : \mathbb{k}'$ *and* $\mathbb{k} \rightsquigarrow^* \mathbb{k}'$.

Proof. As usual for Subject Reduction theorems we proceed by case on the reduction rule used in $cn \to cn'$. Since our types are behavioral we cannot assume that the types of a runtime programs state and its reduction are equivalent, instead we prove that this types $(\mathbb{k}, \mathbb{k}')$ are related according to the operational semantics for runtime types, see Figure A.12. We present the proof of some cases, the rest are either straightforward or similar to those below. Whenever the construction of ι is omitted it is considered as the identity function. Finally, if not stated otherwise, we implicitly use the notation used in the considered reduction rule:

- Case (Assign).

$$\frac{\text{(Assign)}\quad v = [\![e]\!]_l}{vm(o, \top, \{l \mid x = e; s\}, q) \to vm(o, \top, \{l[x \mapsto v] \mid s\}, q)}$$

By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e; s\}, q) : \Bbbk$. By application of the corresponding typing rules we have that $\Bbbk = vm(o, \top \mid (f, 0 \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}})$. Let $\Gamma' = \Gamma$, then by application of rules (BT-VM) and (BT-Process), and by hypothesis we have that $\Gamma' \vdash vm(o, \top, \{l[x \mapsto v] \mid s\}, q) : \Bbbk'$, with $\Bbbk' = vm(o, \top \mid (f, \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}})$. We notice that $\Bbbk$ and $\Bbbk'$ are related by rule (R-Skip)

- Case (Read-Fut).

$$\frac{f = [\![e]\!]_l \quad v \neq \bot}{vm(o, \top, \{l \mid x = e.\mathtt{get}; s\}, q) \quad fut(f, v) \to vm(o, \top, \{l \mid x = v; s\}, q) \quad fut(f, v)} \text{(Read-Fut)}$$

By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e.\mathtt{get}; s\}, q) \quad fut(f, v) : \Bbbk$. Let consider the case in which $\Gamma(f) = (\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})$ (the other case, where $\Gamma(f) = \mathbb{o}$, is trivial). By reconstruction of the proof tree of the hypothesis, with $l = \{x_i \mapsto \mathbb{x}_i \mid i \in I\}$, we derive that $\Bbbk = vm(o, \top \mid (f', f^{\checkmark}_{\mathbb{o}} \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\mathbb{o}'')$ and that $\Gamma[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'][\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}'}[f \mapsto \mathbb{o}] \vdash s : \mathbb{c}$ with $\mathtt{R}' = fv(\mathbb{o}) \setminus \mathtt{R}$ and $\mathbb{t}' = \mathbb{t} \sqcap (\Gamma(\alpha)\Downarrow)$.

Let $\iota = [^{\mathbb{o}''}/_{\mathbb{o}}]$: we have that $\iota(\Gamma) = \Gamma$. Let $\Gamma' = \Gamma[f \mapsto \mathbb{o}'']$, by lemma Appendix A.1 we have that $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'][\beta \mapsto \varnothing\bot]^{\beta \in \iota(\mathtt{R})}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}''} \vdash s : \iota(\mathbb{c})$ with $\mathtt{R}'' = fv(\mathbb{o}'') \setminus \iota(\mathtt{R})$. To prove the theorem in this case, we need to show that s is typable with the environment $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}']$. First, because of the side condition in rule (BT-Fut-Unsync), we have that $\Gamma'(\beta) = \varnothing\bot$ for all $\beta \in \iota(\mathtt{R})$. We thus have

$$\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'][\beta \mapsto \varnothing\bot]^{\beta \in \iota(\mathtt{R})}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}''} = \Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'][\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}''}$$

Now, to remove the last substitution $[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}''}$, we have four cases:

- either $\mathtt{R}''$ is empty, in which case that substitution is the identity;
- either $\mathbb{t}'$ is $\top$, in which case, by construction of the set $\mathtt{R}$, we have $\Gamma'(\beta') = \Gamma(\beta) = \top$: the substitution does not change Γ';
- either $\mathbb{t}'$ is ∂ and $\Gamma'(\beta')$ is $\top$, in which case $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash s : \iota(\mathbb{c})$ holds;
- either $\mathbb{t}'$ is ∂ and $\Gamma'(\beta')$ is $\bot$, in which case $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash s : \mathbb{c}'$ holds, with $\iota(\mathbb{c})$ and $\mathbb{c}'$ being related with few (possibly none) application of the rule (R-Release-GC): statements $\mathtt{release}\ \beta$ are now typed with (T-Release-Bot) instead of (T-Release).

We can then conclude that $\Gamma' \vdash vm(o, \top, \{l \mid x = v; s\}, q) \quad fut(f, v) : \Bbbk'$ with $\Bbbk' = vm(o, \top \mid (f', \mathbb{c}', \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\mathbb{o}'')$, and behavioral types $\Bbbk$ and $\Bbbk'$ are related by (R-Read-Fut), and possibly few application of (R-Release-GC).

- Case (Async-Call).

$$\frac{o' = [\![e]\!]_l \quad \overline{v} = [\![\overline{e}]\!]_l \quad f = \text{fresh}(\,)}{vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) \to vm(o, \top, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v})} \text{(Async-Call)}$$

Lets consider the case when $\Gamma(o') = \top$ (the opposite case $\Gamma(o') = \bot$ is almost identical). By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) : \Bbbk$. By reconstruction we have that $\Bbbk = vm(o, \top \mid (f', \nu f'' : \mathtt{m}\ o'(\overline{\mathbb{s}}) \fatsemi \mathbb{c}, \mathtt{S}'),\ \overline{\mathbb{f}})$ and that there is Γ'' such that $\Gamma[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash\ !\mathtt{m}(\overline{e}) : \nu f'' : \mathtt{m}\ o'(\overline{\mathbb{s}}) \triangleright \Gamma''$ and $\Gamma'' \vdash s : \mathbb{c}$.

Let $\iota = [^f/_{f''}]$ and lets choose Γ' such that $\Gamma'' = \Gamma'[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}']$, by lemma Appendix A.1 we have that $\iota(\Gamma')[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) \to vm(o, \top, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v}) : \Bbbk'$ with $\Bbbk' = vm(o, \top \mid (f', 0 \fatsemi \mathbb{c}', \mathtt{S}'),\ \overline{\mathbb{f}}) \parallel (f'', \mathtt{m}\ o'(\overline{\mathbb{r}}))$, where $\mathbb{c}' = \iota(\mathbb{c})$. Behavioral types $\Bbbk$ and $\Bbbk'$ are related by (R-Async-Call).

- Case (Release-VM).

$$\frac{o' = [\![e]\!]_l \quad o \neq o'}{vm(o, \top, \{l \mid \texttt{release}\ e; s\}, q)\ vm(o', a', p', q') \rightarrow vm(o, \top, \{l \mid s\}, q)\ \ vm(o', \bot, p', q')}\ (\text{Release-VM})$$

By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid \texttt{release}\ e; s\}, q)\ vm(o', a', p', q') : \Bbbk$ and we can derive that $\Bbbk = vm(o, a \mid (f, o'^{\checkmark} \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel vm(o', a' \mid \overline{\mathbb{f}'})$. Let $\Gamma' = \Gamma[o' \mapsto \varnothing\bot]$, we have that $\Gamma' \vdash vm(o, \top, \{l \mid s\}, q)\ vm(o', \bot, p', q') : \Bbbk'$ with $\Bbbk' = vm(o, a \mid (f, \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel vm(o', \bot \mid \overline{\mathbb{f}'})$. Behavioral types $\Bbbk$ and $\Bbbk'$ are related by (R-Release-VM)

- Case (Return).

$$\frac{v = [\![e]\!]_l \quad f = l(\text{destiny})}{vm(o, \top, \{l \mid \texttt{return}\ e\}, q) \quad fut(f, \bot) \rightarrow vm(o, \top, \{l \mid \varepsilon\}, q) \quad fut(f, v)}\ (\text{Return})$$

By reconstruction of the proof tree of the hypothesis after the application of rules (BT-Parallel), (BT-Vm), (BT-Process) and (BT-Return), we can derive that $\Bbbk = vm(o, \top \mid (f, [\mathbb{o}], \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\bot)$. Let $\Gamma' = \Gamma$, by hypothesis and by rule (BT-Fut-Unsync) we have that $\Gamma' \vdash vm(o, \top, \{l \mid \varepsilon\}, p_2\ \ldots\ p_n)\ \ fut(f, v) : \Bbbk'$ with $\Bbbk' = vm(o, \top \mid (f, \mathtt{0} \triangleright \varnothing, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\mathbb{o})$. Note that the side condition $\alpha \in \mathtt{R}[{}^{\mathbb{o}'}/_{\mathbb{o}}] \Rightarrow \Gamma(\alpha) = \varnothing\bot$ of the rule (BT-Fut-Unsync) is a consequence of the constraints $\mathtt{R} = (\mathtt{S} \cup fv(\mathbb{o})) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}$ of the (BT-Process) rule. Finally, the behavioral types $\Bbbk$ and $\Bbbk'$ are related by (R-Return).

- Case (VM-Err-Return).

$$\frac{f = l(\text{destiny})}{\begin{array}{l} vm(o, \bot, \{l \mid s\}, q)\ fut(f, \bot) \\ \rightarrow vm(o, \bot, \{l; \varepsilon\}, q)\ fut(f, err) \end{array}}\ (\text{VM-Err-Return})$$

Let $\Gamma \vdash vm(o, \bot, \{l \mid s\}, q)\ fut(f, \bot) : \Bbbk$ where $\Bbbk = vm(o, \bot \mid (f, \mathtt{0} \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\bot)$. Let $\Gamma' = \Gamma$, the typing of left and right configurations produces an almost identical derivation tree, hence $\Gamma' \vdash vm(o, \bot, \{l \mid \varepsilon\}, q)\ fut(f, err) : \Bbbk'$, and $\Bbbk' = vm(o, \bot \mid (f, \mathtt{0} \triangleright \varnothing, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel fut(f, err)$, which are related by rule (R-Release-Bot).

□

Appendix A.2. Cost Computation

The second part of the correctness intends to demonstrate that, given a computation, the costs of configurations therein do not increase. This means that the cost of the first configuration is the greatest one and, therefore, the cost of the program gives an upper bound of the actual cost of every computation of its.

In order to prove this property, we need to extend the notion of cost to runtime types, and show two properties of these cost: i) they are an upper bound for the number of alive vm in the typed configuration; and ii) they decrease when the configuration executes, thus showing that the cost computed for the initial program is an upper bound for the number of alive vm at every step of its execution.

Cost of Runtime Types. In this paragraph, we extend the definition ofthe `translate` function to runtime type $\Bbbk$. we first introduce our extension with some useful getters on *well-formed* runtime types (i.e., types corresponding to running configurations)

Definition Appendix A.4. *A runtime type $\Bbbk$ is* well-formed *iff all the names (object, future, etc) it uses are declared and if it contains the object: $vm(start, a \mid (f_{start}, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}})$. Given a well-formed runtime type $\Bbbk$, we write:* $\mathtt{alive}(\Bbbk)$ *the number of alive vm in $\Bbbk$ (i.e., the vm whose state is $\top$ in $\Bbbk$);* $\mathtt{mapping}(\Bbbk)$ *the mapping giving the state of vm in $\Bbbk$; and $\Delta_{\Bbbk}$ the set $\{\{o\} \mid o$ is an object of $\Bbbk\}$ (i.e., $\Delta_{\Bbbk}$ is the identity equivalence relation generated from $\Bbbk$).*

We extend the function `translate` as presented in Figure A.13 and define:

Extension of `translate` on extended atoms:

$$\texttt{translate}[\Delta,\Psi,\alpha](\overline{e};e)(\mathbb{a}) = \begin{cases} (\Psi,\ \overline{e};e) & \text{when } \mathbb{a} = [o] \\ (\Psi,(\overline{e};e)\sigma;(e)\sigma') & \text{when } \mathbb{a} = f_{\mathbb{o}}{}^{\checkmark} \text{ and } f \notin \mathrm{dom}(\Psi), \text{ where } \sigma = [f_{\text{peak}}/f] \text{ and } \sigma' = [f_{\text{net}}/f] \\ (\Psi',(\overline{e};e)\sigma;(e)\sigma') & \text{when } \Psi(f) = (\Psi',\mathtt{R},\mathtt{m}\,\beta(\overline{\mathtt{r}},\overline{\beta}) \to \mathbb{o}) \text{ and } \mathbb{a} = f_{\mathbb{o}}{}^{\checkmark} \\ & \text{and } \texttt{EqRel}(\beta,\overline{\beta}) = \Xi \text{ and } \Theta = \Psi' \searrow^{f} \Psi \\ & \text{and } \sigma = [\mathtt{m}^{\Xi}_{\text{peak}}(\overline{\mathtt{r}},\Theta(\Xi(\beta,\overline{\beta})))/f] \ \text{ and } \ \sigma' = [\mathtt{m}^{\Xi}_{\text{net}}(\overline{\mathtt{r}},\Theta(\Xi(\beta,\overline{\beta})))/f] \end{cases}$$

Extension of `translate` on process types:

$$\texttt{translate}(\Delta,\Psi,o,(f,\mathbb{c},\mathrm{S})) = \begin{cases} f_{\text{peak}} = e_{1,1} & [se_1] \\ \vdots & \\ f_{\text{peak}} = e_{h_n,n} & [se_n] \\ f_{\text{net}} = f_{\text{peak}} & [\alpha = \partial] \\ f_{\text{net}} = e_{h_1,1} & [se_1 \wedge \alpha = \top] \\ \vdots & \\ f_{\text{net}} = e_{h_n,n} & [se_n \wedge \alpha = \top] \end{cases} \quad \begin{array}{l} \text{with } \texttt{translate}(\Delta,\Psi[o \mapsto \alpha],(\texttt{true})\,0,\mathbb{c}) = \\ \bigcup_{i=1}^{n}(se_i)\{e_{1,i};\ldots;e_{h_i,i}\} \end{array}$$

Extension of `translate` on runtime types:

$$\texttt{translate}(\Delta,\Psi,f(\mathbb{o})) = \{f_{\text{peak}} = 0, f_{\text{net}} = 0\} \qquad \texttt{translate}(\Delta,\Psi,vm(o,\top \mid \mathbb{f}_1;\mathbb{f}_2\ldots,\mathbb{f}_n)) = \bigcup_{i\in 1..n} \texttt{translate}(\Delta,\Psi,o,\mathbb{f}_i)$$

$$\texttt{translate}(\Delta,\Psi,vm(o,\bot \mid \mathbb{f};\overline{\mathbb{f}})) = \varnothing \qquad \frac{\Xi = \texttt{EqRel}(\overline{v}) \quad \overline{v'} = \Psi(\Xi(\overline{v}))}{\texttt{translate}(\Delta,\Psi,(f,o\,\mathtt{m}(\overline{v}))) = \begin{cases} f_{\text{peak}} = \mathtt{m}^{\Xi}_{\text{peak}}(\overline{v'}) \\ f_{\text{net}} = \mathtt{m}^{\Xi}_{\text{net}}(\overline{v'}) \end{cases}}$$

$$\texttt{translate}(\Delta,\Psi,\mathbb{k} \parallel \mathbb{k}') = \texttt{translate}(\Delta,\Psi,\mathbb{k}) \cup \texttt{translate}(\Delta,\Psi,\mathbb{k}')$$

Figure A.13: Extension of `translate` to runtime types

Definition Appendix A.5. *The cost equations corresponding to a well-formed runtime type* $\mathbb{k}$*, written* `translate`$(\mathbb{k})$ *is defined as follows:*

$$\texttt{translate}(\mathbb{k}) \stackrel{def}{=} \texttt{translate}(\Delta_{\mathbb{k}},\texttt{mapping}(\mathbb{k}),\mathbb{k})$$

Note in Figure A.13 that we also need to extend the definition of `translate` on atoms: we need to re-define the cost of a `get` operation as we now have not only the actual futures in the runtime type but also the ones coming from the static typing of the program (the ones that belong to Ψ). Moreover the initial definition of `translate` for the `get` included an extra adjustment derived from the fact that the state of some virtual machine were unknown, which is not the case at runtime.

Cost Solution. A *cost solution* Σ is a mapping from cost functions (like $\mathtt{m}^{\Xi}_{\text{net}}(\overline{v})$ or f_{peak}, where f is a future name) to cost expressions. Basically, this cost solution gives cost expressions of the peak and net cost of all methods in the program, as well as, the peak and net cost of all processes in the runtime type. Following [13, Definition 6], we relate cost solutions and runtime types by assuming the existence of a *cost validation predicate* $\vDash$ between cost solutions and cost equations as generated by the `translate` function in Section 5:

Definition Appendix A.6. *Let C be a set of cost equations* $\bigcup_{i=1}^{n} f_i(\overline{x_i}) = e_i[se_i]$*; let* Σ *be a function that maps all* $f_i(\overline{x_i})$ *to cost expressions.* Σ *is a* cost solution *of C, noted* $\Sigma \vDash C$*, if for all* $1 \le i \le n$ *and all substitutions* $[\overline{v}/\overline{x_i}]$ *such that* $se_i[\overline{v}/\overline{x_i}]$ *is true, we have that*

$$\Sigma(f_i)(\overline{v}) \ge \Sigma(e_i[\overline{v}/\overline{x_i}])$$

where $\Sigma(e)$ *replaces the function calls in e by their value in* Σ*.*

Correctness of the Cost Analysis. The proof of the correction of our cost analysis is done in four steps. First, we relate the cost equation computed at stating time with the runtime version of the types:

Lemma Appendix A.7. *Let $\overline{M}$ $\{\overline{F\ z\ ;}\ s'\}$ be a well-typed program, let $\overline{\mathbb{C}}$, $\mathbb{c}$ be its behavioural type, cn be its initial configuration and $\mathbb{k}$ the runtime type of cn. The for all cost solution Σ with $\Sigma \vDash \texttt{translate}(\overline{\mathbb{C}}, \mathbb{c})$, there exists a cost solution Σ' such that $\Sigma' \vDash \texttt{translate}(\mathbb{k})$ and $\Sigma(\texttt{main}()) = \Sigma'(f_{start_{peak}}) + \texttt{alive}(\mathbb{k})$.*

Proof. Let define $\Sigma' = \Sigma[f_{start_{peak}} \mapsto \Sigma(\texttt{main}()) - 1]$. By construction (see the construction of the equation page 15 and Figure A.13), we have that

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

$$\text{with } \texttt{translate}(\varnothing, \varnothing, \alpha, (\texttt{true})\{\,0\,\}, \mathbb{c}) = \bigcup_{j=1}^{m} (se_j)\{\,e_{1,j}; \cdots ; e_{h_j,j}\,\}$$

Hence, by construction, we have that $\Sigma' \vDash \texttt{translate}(\mathbb{k})$. Moreover, as $\texttt{alive}(\mathbb{k}) = 1$, we also have that $\Sigma(\texttt{main}()) = \Sigma'(f_{start_{peak}}) + \texttt{alive}(\mathbb{k})$. □

Second, we show that any solution of the cost equation generated from a runtime type is an upper bound for the number of live vm in that runtime type:

Lemma Appendix A.8. *Let $\Gamma \vdash cn : \mathbb{k}$ and assume there is a cost solution Σ such that $\Sigma \vDash \texttt{translate}(\mathbb{k})$. Then $\texttt{alive}(cn) \leq \texttt{alive}(\mathbb{k}) + \Sigma(f_{start_{peak}})$ (recall that $f_{start_{peak}}$ is the peak cost of the main process type of $\mathbb{k}$).*

Proof. Observe that $\texttt{alive}(cn) = \texttt{alive}(\mathbb{k})$ by construction of $\mathbb{k}$. We moreover notice that $\Sigma(f_{start_{peak}})$ is positive, since peak costs are by construction always positive integers. We thus have the result. □

Third, we show that an upper bound computed by our technique is stable w.r.t. runtime type evolution:

Lemma Appendix A.9. *Suppose given two well-formed runtime types $\mathbb{k}$ and $\mathbb{k}'$ such that $\mathbb{k} \rightsquigarrow \mathbb{k}'$. Then, for all cost solution Σ with $\Sigma \vDash \texttt{translate}(\mathbb{k})$, there exists Σ' such that: i) $\Sigma' \vDash \texttt{translate}(\mathbb{k}')$; and ii) $\texttt{alive}(\mathbb{k}') + \Sigma'(f_{start_{peak}}) \leq \texttt{alive}(\mathbb{k}) + \Sigma(f_{start_{peak}})$.*

Proof. By Case on the rule used in $\mathbb{k} \rightsquigarrow \mathbb{k}'$. If not stated otherwise, we implicitly use the notation used in the considered reduction rule):

- Cases (R-Skip), (R-Read-Fut-Known), (R-Read-Fut-Unknown), (R-Choice), (R-GC), (R-Return), (R-Async-Call-Err). Because $\texttt{alive}(\mathbb{k}) = \texttt{alive}(\mathbb{k}')$, it is clear that taking $\Sigma' = \Sigma$, we have and $\Sigma(f_{start_{peak}}) = \Sigma'(f_{start_{peak}})$. Hence, we have the result.
- Case (R-Async-Call). It is easy to remark that $\texttt{translate}(\mathbb{k}')$ differs from $\texttt{translate}(\mathbb{k})$ in the following points: i) the entries $[f'_{\text{peak}} = \mathtt{m}^{\Xi}_{\text{peak}}(\overline{v'}); f'_{\text{net}} = \mathtt{m}^{\Xi}_{\text{net}}(\overline{v'})]$ (with $\overline{v'} = \Psi(\Xi(\overline{v}))$ and $\Xi = \texttt{EqRel}(\overline{v'})$) have been added to $\texttt{translate}(\mathbb{k}')$; and ii) some references to $\mathtt{m}^{\Xi}_{\text{peak}}(\overline{v'})$ and $\mathtt{m}^{\Xi}_{\text{net}}(\overline{v'})$ in $\texttt{translate}(\mathbb{k})$ have been replaced in $\texttt{translate}(\mathbb{k}')$ by references to f'_{peak} and f'_{net}. So if we define Σ' as follow, we have the result (as $\texttt{alive}(\mathbb{k}) = \texttt{alive}(\mathbb{k}')$):
$$\Sigma' = \Sigma[f'_{\text{peak}} \mapsto \Sigma(\mathtt{m}^{\Xi}_{\text{peak}}(\overline{v'})); f'_{\text{net}} \mapsto \Sigma(\mathtt{m}^{\Xi}_{\text{net}}(\overline{v'}))]$$
- Case (R-Bind-Mtd). Given that $\mathbb{c}$ is the behavior of $\mathtt{m}\ o(\overline{v})$ and that $\texttt{translate}(\mathbb{k}_1 \parallel \mathbb{k}_2) = \texttt{translate}(\mathbb{k}_1) \cup \texttt{translate}(\mathbb{k}_2)$, $\texttt{translate}(\mathbb{k})$ and $\texttt{translate}(\mathbb{k}')$ produce the same set of equations up to the substition of formal parameters. By definition of Σ we have the result for $\Sigma' = \Sigma$.

- Case (R-New-VM). Let f_i be all possible branches of f. We define

$$\Sigma' = \Sigma[f_{i,\text{peak}} \mapsto \Sigma(f_{i\_\text{peak}}) - 1][f_{i\_\text{net}} \mapsto \Sigma(f_{i\_\text{net}}) - 1]$$

  The definition of Σ' corresponds to the tranfer of the cost +1 to the set of alive virtual machines. Hence we $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk') - 1$ and $\Sigma'(f_{start_{peak}}) = \Sigma(f_{start_{peak}}) - 1$, which gives us the result.

- Case (R-Release-VM). If $a' = \top$ we have that $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk') + 1$, if $a' = \bot$ we have that $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk')$, thus taking $\Sigma' = \Sigma$, we have the result.

□

Finally, we can combine all the previous lemma to demonstrate the correctness of our technique.

Theorem 5.1 (Correctness)
Let $\overline{M}$ $\{\overline{F\ z\ ;}\ s'\}$ *be a well-typed program, let* $\overline{\mathbb{C}}$, $\mathbb{c}$ *be its behavioural type and cn be its initial configuration. Let also* Σ *be a solution of the cost functions* $\texttt{translate}(\overline{\mathbb{C}}, \mathbb{c})$. *Then, for every* $cn \longrightarrow^* cn'$, $\texttt{alive}(cn') \leq \Sigma(\texttt{main}())$.

Proof. Let $n = \Sigma(\texttt{main}())$.
The argument proceeds by induction on the number of reduction steps:

- At step 0, we translate the program into its runtime equivalent *cn*. By Lemma Appendix A.7, there exists Σ' such that $\Sigma' \vDash \texttt{translate}(\Bbbk)$ and that we have that $\Sigma'(f_{start_{peak}}) + \texttt{alive}(\Bbbk) = n$. Morever, by Lemma Appendix A.8 we have that $\texttt{alive}(cn) \leq \texttt{alive}(\Bbbk) + \Sigma'(f_{start_{peak}}) = n$.

- for the inductive case, let Γ, a runtime configuration *cn'*, a runtime type $\Bbbk$ and a cost solution Σ' such that $\Gamma \vdash cn : \Bbbk$, $\Sigma' \vDash \Bbbk$ and $\Sigma'(f_{start_{peak}}) + \texttt{alive}(\Bbbk) \leq n$ hold. Then by Theorem Appendix A.3, if $cn \to cn'$, there exists Γ' and a runtime type $\Bbbk'$ such that $\Bbbk \rightsquigarrow^* \Bbbk'$ and $\Gamma' \vdash cn' : \Bbbk'$. By Lemma Appendix A.9, there exist a cost solution Σ'' such that $\Sigma'' \vDash \Bbbk'$ and $\texttt{alive}(\Bbbk') + \Sigma''(f_{start_{peak}}) \leq \texttt{alive}(\Bbbk) + \Sigma'(f_{start_{peak}})$. By hypothesis and Lemma Appendix A.8, we that $\texttt{alive}(cn') \leq \texttt{alive}(\Bbbk') + \Sigma''(f_{start_{peak}}) \leq n$.

□