Time complexity of concurrent programs[⋆]
– a technique based on behavioural types –

Elena Giachino[1], Einar Broch Johnsen[2], Cosimo Laneve[1], and Ka I Pun[2]

[1] Dept. of Computer Science and Engineering, University of Bologna – INRIA FOCUS
[2] Dept. of Informatics, University of Oslo

Abstract. We study the problem of automatically computing the time complexity of concurrent object-oriented programs. To determine this complexity we use intermediate abstract descriptions that record relevant information for the time analysis (cost of statements, creations of objects, and concurrent operations), called *behavioural types*. Then, we define a translation function that takes behavioural types and makes the parallelism explicit into so-called *cost equations*, which are fed to an automatic off-the-shelf solver for obtaining the time complexity.

1 Introduction

Computing the cost of a sequential algorithm has always been a primary question for every programmer, who learns the basic techniques in the first years of their computer science or engineering curriculum. This cost is defined in terms of the input values to the algorithm and over-approximates the number of the executed instructions. In turn, given an appropriate abstraction of the CPU speed of a runtime system, one obtain the expected computation time of the algorithm.

The computational cost of algorithms is particularly relevant in mainstream architectures, such as the cloud. In that context, a service is a concurrent program that must comply with a so-called *service-level agreement* (SLA) regulating the cost in time and assigning penalties for its infringement [?]. The service provider needs to make sure that the service is able to meet the SLA, for example in terms of the end-user response time, by deciding on a resource management policy and determining the appropriate number of virtual machine instances (or containers) and their parameter settings (e.g., their CPU speeds). To help service providers make correct decisions about the resource management before actually deploying the service, we need static analysis methods for resource-aware services [?]. In previous work by the authors, cloud deployments expressed in the formal modeling language ABS [?] have used a combination of cost analysis and simulations to analyse resource management [?], and a Hoare-style proof system to reason about end-user deadlines has been developed for sequential executions [?]. In contrast, we are here interested in statically estimating the

⋆ Supported by the EU projects FP7-610582 Envisage: Engineering Virtualized Services (`http://www.envisage-project.eu`).

computation time of concurrent services deployed on the cloud with a given dynamic resource management policy.

Technically, this paper proposes a behavioural type system expressing the resource costs associated with computations and study how these types can be used to soundly calculate the time complexity of parallel programs deployed on the cloud. To succinctly formulate this problem, our work is developed for `tml`, a small formally defined concurrent object-oriented language which uses asynchronous communications to trigger parallel activities. The language defines virtual machine instances in terms of dynamically created concurrent object groups with bounds on the number of cycles they can perform per time interval. As we are interested in the concurrent aspects of these computations, we abstract from sequential analysis in terms of a statement $\texttt{job}(e)$, which defines the number of processing cycles required by the instruction – this is similar to the `sleep(n)` operation in `Java`.

The analysis of behavioural types is defined by translating them in a code that is adequate for an off-the-shelf solver – the `CoFloCo` solver [?]. As a consequence, we are able to determine the computational cost of algorithms in a parametric way with respect to their inputs.

Paper overview. The language is defined in Section 2 and we discuss restrictions that ease the development of our technique in Section 3. Section 4 presents the behavioural type system and Section 5 explains the analysis of computation time based on these behavioural types. In Section 6 we outline our correctness proof of the type system with respect to the cost equations. In Section 7 we discuss the relevant related work and in Section 8 we deliver concluding remarks.

2 The language `tml`

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

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

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

$$
\begin{array}{l}
s ::= x = z \ \mid\ \texttt{if}\ e\ \{\,s\,\}\ \texttt{else}\ \{\,s\,\} \ \mid\ \texttt{job}(e) \ \mid\ \texttt{return}\ e \ \mid\ s\ ;\ s \\
z ::= e \ \mid\ e\texttt{!m}(\overline{e}) \ \mid\ e.\texttt{m}(\overline{x}) \ \mid\ e.\texttt{get} \ \mid\ \texttt{new Class with}\ e \ \mid\ \texttt{new local Class} \\
e ::= \texttt{this} \ \mid\ se \ \mid\ nse
\end{array}
$$

A statement s may be either one of the standard operations of an imperative language or the job statement $\texttt{job}(e)$ that delays the continuation by e cycles of the machine executing it.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method invocation of the form $e\texttt{!m}(\overline{e})$, which does not suspend the caller's execution. When the value computed by the invocation is needed, 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, i.e., *await-get*. Expressions z also include standard synchronous invocations $e.\texttt{m}(\overline{e})$ and `new local Class`, which creates a new object. The intended meaning is to create the object in the same machine – called *cog* or *concurrent object group* – of the caller, thus sharing the processor of the caller: operations in the same virtual machine interleave their evaluation (even if in the following operational semantics the parallelism is not explicit). Alternatively, one can create an object on a different cog with `new Class with` e thus letting methods execute in parallel. In this case, e represents the capacity of the new cog, that is, the number of cycles the cog can perform per time interval. We assume the presence of a special identifier `this.capacity` that returns the capacity of the corresponding cog.

A *pure* expression e can be the reserved identifier `this` or an integer expression. Since the analysis in Section 5 cannot deal with generic integer expressions, we parse expressions in a careful way. In particular we split them into *size expressions se*, which are expressions in Presburger arithmetics (this is a decidable fragment of Peano arithmetics that only contains addition), and *non-size expressions nse*, which are the other type of expressions. The syntax of size and non-size expressions is the following:

$$\begin{array}{rl}
\mathit{nse} ::= & k \;\mid\; x \;\mid\; \mathit{nse} \leq \mathit{nse} \;\mid\; \mathit{nse}\ \texttt{and}\ \mathit{nse} \;\mid\; \mathit{nse}\ \texttt{or}\ \mathit{nse} \\
& \mid\; \mathit{nse} + \mathit{nse} \;\mid\; \mathit{nse} - \mathit{nse} \;\mid\; \mathit{nse} \times \mathit{nse} \;\mid\; \mathit{nse}/\mathit{nse} \\
\mathit{se} ::= & \mathit{ve} \;\mid\; \mathit{ve} \leq \mathit{ve} \;\mid\; \mathit{se}\ \texttt{and}\ \mathit{se} \;\mid\; \mathit{se}\ \texttt{or}\ \mathit{se} \\
\mathit{ve} ::= & k \;\mid\; x \;\mid\; \mathit{ve} + \mathit{ve} \;\mid\; k \times \mathit{ve} \\
k ::= & \mathit{rational\ constants}
\end{array}$$

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

Semantics. The semantics of `tml` is defined by a transition system whose states are *configurations cn* that are defined by the following syntax.

$$\begin{array}{rlcrl}
\mathit{cn} ::= & \varepsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, c, p, q) \mid \mathit{invoc}(o, f, \texttt{m}, \overline{v}) & \quad & \mathit{act} ::= & o \mid \varepsilon \\
\mid & \mathit{cog}(c, \mathit{act}, k) \mid \mathit{cn}\ \mathit{cn} & & \mathit{val} ::= & v \mid \bot \\
p ::= & \{\, l \mid s \,\} \mid \texttt{idle} & & l ::= & [\cdots, x \mapsto v, \cdots] \\
q ::= & \varnothing \mid \{\, l \mid s \,\} \mid q\ q & & v ::= & o \mid f \mid k
\end{array}$$

A *configuration cn* is a set of concurrent object groups (cogs), objects, invocation messages and futures, and the empty configuration is written as ε. The associative and commutative union operator on configurations is denoted by whitespace. A *cog* is given as a term $\mathit{cog}(c, \mathit{act}, k)$ where c and k are respectively the identifier and the capacity of the cog, and *act* specifies the currently active

$$\textsc{(Cond-True)}\quad \dfrac{\texttt{true} = \llbracket e\rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2\,\}\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid s_1\ \texttt{;}\ s\,\},q)\end{array}}$$

$$\textsc{(Cond-False)}\quad \dfrac{\texttt{false} = \llbracket e\rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2\,\}\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid s_2\ \texttt{;}\ s\,\},q)\end{array}}$$

$$\textsc{(New)}\quad \dfrac{c' = \mathrm{fresh}(\,)\quad o' = \mathrm{fresh}(\,)\quad k = \llbracket e\rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = \texttt{new Class with}\ e\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid x = o'\ \texttt{;}\ s\,\},q) \\ ob(o',c',\texttt{idle},\varnothing)\quad cog(c',o',k)\end{array}}$$

$$\textsc{(New-Local)}\quad \dfrac{o' = \mathrm{fresh}(\,)}{\begin{array}{c} ob(o,c,\{\, l \mid x = \texttt{new local Class}\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid x = o'\ \texttt{;}\ s\,\},q) \\ ob(o',c,\texttt{idle},\varnothing)\end{array}}$$

$$\textsc{(Get-True)}\quad \dfrac{f = \llbracket e\rrbracket_l\quad v \neq \bot}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s\,\},q)\quad fut(f,v) \\ \to ob(o,c,\{\, l \mid x = v\ \texttt{;}\ s\,\},q)\quad fut(f,v)\end{array}}$$

$$\textsc{(Get-False)}\quad \dfrac{f = \llbracket e\rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s\,\},q)\quad fut(f,\bot) \\ \to ob(o,c,\texttt{idle},q \cup \{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s\,\})\quad fut(f,\bot)\end{array}}$$

$$\textsc{(Self-Sync-Call)}\quad \dfrac{\begin{array}{c} o = \llbracket e\rrbracket_l\quad \overline{v} = \llbracket \overline{e}\rrbracket_l\quad f' = l(\texttt{destiny}) \\ f = \mathrm{fresh}(\,)\quad \{\, l' \mid s'\,\} = \mathrm{bind}(o,f,\texttt{m},\overline{v})\end{array}}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{m}(\overline{e})\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l' \mid s'\ \texttt{;}\ \texttt{cont}(f')\,\},q \cup \{\, l \mid x = f.\texttt{get}\ \texttt{;}\ s\,\}) \\ fut(f,\bot)\end{array}}$$

$$\textsc{(Self-Sync-Return-Sched)}\quad \dfrac{f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{cont}(f)\,\},q \cup \{\, l' \mid s\,\}) \\ \to ob(o,c,\{\, l' \mid s\,\},q)\end{array}}$$

$$\textsc{(Cog-Sync-Call)}\quad \dfrac{\begin{array}{c} o' = \llbracket e\rrbracket_l\quad \overline{v} = \llbracket \overline{e}\rrbracket_l\quad f' = l(\texttt{destiny}) \\ f = \mathrm{fresh}(\,)\quad \{\, l' \mid s'\,\} = \mathrm{bind}(o',f,\texttt{m},\overline{v})\end{array}}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{m}(\overline{e})\ \texttt{;}\ s\,\},q) \\ ob(o',c,\texttt{idle},q')\quad cog(c,o,k) \\ \to ob(o,c,\texttt{idle},q \cup \{\, l \mid x = f.\texttt{get}\ \texttt{;}\ s\,\})\quad fut(f,\bot) \\ ob(o',c,\{\, l' \mid s'\ \texttt{;}\ \texttt{cont}(f')\,\},q')\quad cog(c,o',k)\end{array}}$$

$$\textsc{(Cog-Sync-Return-Sched)}\quad \dfrac{f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{cont}(f)\,\},q)\quad cog(c,o,k) \\ ob(o',c,\texttt{idle},q' \cup \{\, l' \mid s'\,\}) \\ \to ob(o,c,\texttt{idle},q)\quad cog(c,o',k) \\ ob(o',c,\{\, l' \mid s'\,\},q')\end{array}}$$

$$\textsc{(Async-Call)}\quad \dfrac{o' = \llbracket e\rrbracket_l\quad \overline{v} = \llbracket \overline{e}\rrbracket_l\quad f = \mathrm{fresh}(\,)}{\begin{array}{c} ob(o,c,\{\, l \mid x = e!\texttt{m}(\overline{e})\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid x = f\ \texttt{;}\ s\,\},q)\quad invoc(o',f,\texttt{m},\overline{v})\quad fut(f,\bot)\end{array}}$$

$$\textsc{(Bind-Mtd)}\quad \dfrac{\{\, l \mid s\,\} = \mathrm{bind}(o,f,\texttt{m},\overline{v})}{\begin{array}{c} ob(o,c,p,q)\quad invoc(o,f,\texttt{m},\overline{v}) \\ \to ob(o,c,p,q \cup \{\, l \mid s\,\})\end{array}}$$

$$\textsc{(Context)}\quad \dfrac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}$$

$$\textsc{(Release-Cog)}\quad \begin{array}{c} ob(o,c,\texttt{idle},q)\quad cog(c,o,k) \\ \to ob(o,c,\texttt{idle},q)\quad cog(c,\varepsilon,k)\end{array}$$

$$\textsc{(Activate)}\quad \begin{array}{c} ob(o,c,\texttt{idle},q \cup \{\, l \mid s\,\})\quad cog(c,\varepsilon,k) \\ \to ob(o,c,\{\, l \mid s\,\},q)\quad cog(c,o,k)\end{array}$$

$$\textsc{(Return)}\quad \dfrac{v = \llbracket e\rrbracket_l\quad f = l(\mathrm{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{return}\ e\,\},q)\quad fut(f,\bot) \\ \to ob(o,c,\texttt{idle},q)\quad fut(f,v)\end{array}}$$

$$\textsc{(Job-0)}\quad \dfrac{\llbracket e\rrbracket_l = 0}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{job}(e)\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l \mid s\,\},q)\end{array}}$$

$$\textsc{(Assign-Local)}\quad \dfrac{x \in \mathrm{dom}(l)\quad v = \llbracket e\rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = e\ \texttt{;}\ s\,\},q) \\ \to ob(o,c,\{\, l\,[x \mapsto v] \mid s\,\},q)\end{array}}$$

Fig. 1. The transition relation of `tml` – part 1.

object in the cog. An object is written as $ob(o, c, p, q)$, where o is the identifier of the object, c the identifier of the cog the object belongs to, p an *active process*, and q a pool of *suspended processes*. A *process* is written as $\{\, l \mid s\,\}$, where l denotes local variable bindings and s a list of statements. An *invocation message* is a term $invoc(o, f, \texttt{m}, \overline{v})$ consisting of the callee o, the future f to which the result of the call is returned, the method name m, and the set of actual parameter values for the call. A *future* $fut(f, val)$ contains an identifier f and a reply value val, where $\bot$ indicates the reply value of the future has not been received.

The following auxiliary function is used in the semantic rules for invocations. Let $T'\ \texttt{m}(\overline{T\ x})\{\,\overline{F\ x'}; s\,\}$ be a method declaration. Then

$$\mathrm{bind}(o, f, \texttt{m}, \overline{v}) = \{\, [\mathrm{destiny} \mapsto f, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \bot] \mid s\{^{o}/_{\texttt{this}}\}\,\}$$

$$\text{(TICK)}\quad \frac{\mathit{strongstable}_t(cn)}{cn \to \Phi(cn,t)}$$

where

$$\Phi(cn,t) \;=\; \begin{cases} ob(o,c,\{l' \mid \mathtt{job}(k')\ ;\ s\},q)\ \Phi(cn',t) & \text{if } cn = ob(o,c,\{l \mid \mathtt{job}(e)\ ;\ s\},q)\ cn' \\ & \text{and } cog(c,o,k) \in cn' \\ & \text{and } k' = [\![e]\!]_l - k * t \\ \\ ob(o,c,\mathtt{idle},q)\ \Phi(cn',t) & \text{if } cn = ob(o,c,\mathtt{idle},q)\ cn' \\ \\ cn & \text{otherwise.} \end{cases}$$

Fig. 2. The transition relation of `tml` – part 2: the strongly stable case

The *transition rules* of `tml` are given in Figures 1 and 2. We discuss the most relevant ones: object creation, method invocation, and the $\mathtt{job}(e)$ operator. The creation of objects is handled by rules New and New-Local: the former creates a new object inside a new cog with a given capacity e, the latter creates an object in the local cog. Method invocations can be either synchronous or asynchronous. Rules Self-Sync-Call and Cog-Sync-Call specify synchronous invocations on objects belonging to the same cog of the caller. Asynchronous invocations can be performed on every object.

In our model, the unique operation that consumes time is $\mathtt{job}(e)$. We notice that the reduction rules of Figure 1 are not defined for the $\mathtt{job}(e)$ statement, except the trivial case when the value of e is 0. This means that time does not advance while non-job statements are evaluated. When the configuration cn reaches a *stable* state, *i.e.,* no other transition is possible apart from those evaluating the $\mathtt{job}(e)$ statements, then the time is advanced by the minimum value that is necessary to let at least *one* process start. In order to formalize this semantics, we define the notion of stability and the *update operation* of a configuration cn (with respect to a time value t). Let $[\![e]\!]_l$ return the value of e when variables are bound to values stored in l.

Definition 1. *Let $t > 0$. A configuration cn is* t-stable*, written $\mathit{stable}_t(cn)$, if any object in cn is in one of the following forms:*

1. $ob(o,c,\{\, l \mid \mathtt{job}(e); s\,\},q)$ *with* $cog(c,o,k) \in cn$ *and* $[\![e]\!]_l/k \geq t$,
2. $ob(o,c,\mathtt{idle},q)$ *and*
   i. *either* $q = \varnothing$,
   ii. *or, for every* $p \in q$, $p = \{\, l \mid x = e.\mathtt{get}; s\,\}$ *with* $[\![e]\!]_l = f$ *and* $fut(f,\bot)$,
   iii. *or,* $cog(c,o',k) \in cn$ *where* $o \neq o'$, *and* o' *satisfies Definition 1.1.*

A configuration cn is strongly t-stable*, written $\mathit{strongstable}_t(cn)$, if it is t-stable and there is an object $ob(o,c,\{\, l \mid \mathtt{job}(e); s\,\},q)$ with $cog(c,o,k) \in cn$ and $[\![e]\!]_l/k \;=\; t$.*

Notice that t-stable (and, consequently, strongly t-stable) configurations cannot progress anymore because every object is stuck either on a job or on unresolved get statements. The update of cn with respect to a time value t, noted $\Phi(cn, t)$ is defined in Figure 2. Given these two notions, rule Tick defines the time progress. The initial configuration of a program with main method $\{\,\overline{F\ x};\ s\,\}$ `with` k is

$$
\begin{array}{l}
ob(start, \mathrm{start}, \{\,[\mathrm{destiny} \mapsto f_{start}, \overline{x} \mapsto \bot] \mid s\,\}, \varnothing) \\
cog(\mathrm{start}, start, k)
\end{array}
$$

where start and *start* are special cog and object names, respectively, and f_{start} is a fresh future name. As usual, $\to^*$ is the reflexive and transitive closure of $\to$.

Examples. To begin with, we discuss the Fibonacci method. It is well known that the computational cost of its sequential recursive implementation is exponential. However, this is not the case for the parallel implementation. Consider

```
Int fib(Int n) {
      if (n<=1) { return 1; }
      else {  Fut<Int> f; Class z; Int m1; Int m2;
              job(1);
              z = new Class with this.capacity ;
              f = this!fib(n-1); g = z!fib(n-2);
              m1 = f.get; m2 = g.get;
              return  m1 + m2; } }
```

Here, the recursive invocation `fib(n-1)` is performed on the `this` object while the invocation `fib(n-2)` is performed on a new cog with the same capacity (i.e., the object referenced by `z` is created in a new cog set up with `this.capacity`), which means that it can be performed in parallel with the former one. It turns out that the cost of the following invocation is `n`.

```
Class z; Int m; Int x; x = 1;
z = new Class with x;
m = z.fib(n);
```

Observe that, by changing the line `x = 1;` into `x = 2;` we obtain a cost of `n/2`.

Our semantics does not exclude paradoxical behaviours of programs that perform infinite actions without consuming time (preventing rule Tick to apply), such as this one

```
Int foo() { Int m; m = this.foo(); return m; }
```

This kind of behaviours are well-known in the literature, (*cf.* Zeno behaviours) and they may be easily excluded from our analysis by constraining recursive invocations to be prefixed by a `job`(e)-statement, with a positive e. It is worth to observe that this condition is not sufficient to eliminate paradoxical behaviours. For instance the method below does not terminate and, when invoked with `this.fake(2)`, where `this` is in a cog of capacity 2, has cost 1.

```
Int fake(Int n) {
     Int m; Class x;
     x = new Class with 2*n; job(1); m = x.fake(2*n); return m; }
```

Imagine a parallel invocation of the method `Int one() { job(1); }` on an object residing in a cog of capacity 1. At each stability point the `job`(1) of the latter method will compete with the `job`(1) of the former one, which will win every time, since having a greater (and growing) capacity it will require always less time. So at the first stability point we get `job`$(1-1/2)$ (for the method `one`), then `job`$(1-1/2-1/4)$ and so on, thus this sum will never reach 0.

In the examples above, the statement `job`(e) is a cost annotation that specifies how many processing cycles are needed by the subsequent statement in the code. We notice that this operation can also be used to program a timer which suspends the current execution for e units of time. For instance, let

```
Int wait(Int n) { job(n); return 0; }
```

Then, invoking `wait` on an object with capacity 1

```
Class timer; Fut<Class> f; Class x;
timer = new Class with 1;
f = timer!wait(5); x = f.get;
```

one gets the suspension of the current thread for 5 units of time.

3 Issues in computing the cost of `tml` programs

The computation time analysis of `tml` programs is demanding. To highlight the difficulties, we discuss a number of methods.

```
Int wrapper(Class x) {
      Fut<Int> f; Int z;
      job(1) ; f = x!server(); z = f.get;
      return z; }
```

Method `wrapper` performs an invocation on its argument `x`. In order to determine the cost of `wrapper`, we notice that, if `x` is in the same cog of the carrier, then its cost is (assume that the capacity of the carrier is `1`): `1`$+$*cost*(`server`) because the two invocations are sequentialized. However, if the cogs of `x` and of the carrier are different, then we are not able to compute the cost because we have no clue about the state of the cog of `x`.

Next consider the following definition of `wrapper`

```
Int wrapper_with_log(Class x) {
      Fut<Int> f; Fut<Int> g; Int z;
      job(1) ; f = x!server(); g = x!print_log(); z = f.get;
      return z; }
```

In this case the wrapper also asks the server to print its log and this invocation is not synchronized. We notice that the cost of `wrapper_with_log` is not anymore `1` $+$ *cost*(`server`) (assuming that `x` is in the same cog of the carrier) because `print_log` might be executed *before* `server`. Therefore the cost of `wrapper_with_log` is $1+$ *cost*(`server`) $+$ *cost*(`print_log`).

Finally, consider the following wrapper that also logs the information received from the server on a new cog without synchronising with it:

```
Int wrapper_with_external_log(Class x) {
      Fut<Int> f; Fut<Int> g; Int z; Class y;
      job(1) ; f = x!server(); g = x!print_log(); z = f.get;
      y = new Class with 1;
      f = y!external_log(z);
      return z; }
```

What is the cost of `wrapper_with_external_log`? Well, the answer here is debatable: one might discard the cost of `y!external_log(z)` because it is useless for the value returned by `wrapper_with_external_log`, or one might count it because one wants to count every computation that has been triggered by a method in its cost. In this paper we adhere to the second alternative; however, we think that a better solution should be to return different cost for a method: a *strict cost*, which spots the cost that is necessary for computing the returned value, and an *overall cost*, which is the one computed in this paper.

Anyway, by the foregoing discussion, as an initial step towards the time analysis of `tml` programs, we simplify our analysis by imposing the following constraint:

– *it is possible to invoke methods on objects either in the same cog of the caller or on newly created cogs.*

The above constraint means that, if the callee of an invocation is one of the arguments of a method then it must be in the same cog of the caller. It also means that, if an invocation is performed on a returned object then this object must be in the same cog of the carrier. We will enforce these constraints in the typing system of the following section – see rule T-Invoke.

4 A behavioural type system for `tml`

In order to analyse the computation time of `tml` programs we use abstract descriptions, called *behavioural types*, which are intermediate codes highlighting the features of `tml` programs that are relevant for the analysis in Section 5. These abstract descriptions support compositional reasoning and are associated to programs by means of a type system. The syntax of behavioural types is defined as follows:

$\mathbb{t} ::=$	$- \mid se \mid c[se]$	basic value
$\mathbb{x} ::=$	$f \mid \mathbb{t}$	extended value
$\mathbb{a} ::=$	$e \mid \nu c[se] \mid \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t} \mid \nu f\colon \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t} \mid f^{\checkmark}$	atom
$\mathbb{b} ::=$	$\mathbb{a} \triangleright \Gamma \mid \mathbb{a} \fatsemi \mathbb{b} \mid (se)\{\, \mathbb{b} \,\} \mid \mathbb{b} + \mathbb{b}$	behavioural type

where c, c', $\cdots$ range over cog names and f, f', $\cdots$ range over future names. Basic values $\mathbb{t}$ are either generic (non-size) expressions $-$ or size expressions se or the type $c[se]$ of an object of cog c with capacity se. The extended values add future names to basic values.

Atoms $\mathbb{a}$ define creation of cogs ($\nu c[se]$), synchronous and asynchronous method invocations ($\mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}$ and $\nu f\colon \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}$, respectively), and synchronizations on asynchronous invocations ($f^{\checkmark}$). We observe that cog creations always carry a capacity, which has to be a size expression because our analysis in the next section cannot deal with generic expressions. Behavioural types $\mathbb{b}$ are sequences of atoms $\mathbb{a} \fatsemi \mathbb{b}'$ or conditionals, typically $(se)\{\, \mathbb{b} \,\} + (\neg se)\{\, \mathbb{b}' \,\}$ or $\mathbb{b} + \mathbb{b}'$, according to whether the boolean guard is a size expression that depends on the arguments of a method or not. In order to type sequential composition in a precise way (see rule T-Seq), the leaves of behavioural types are labelled with *environments*, ranged over by Γ, Γ', $\cdots$. Environments are maps from method names $\mathtt{m}$ to terms $(\overline{\mathbb{t}}) \to \mathbb{t}$, from variables to extended values $\mathbb{x}$, and from future names to values that are either $\mathbb{t}$ or $\mathbb{t}^{\checkmark}$.

The abstract behaviour of methods is defined by *method behavioural types* of the form: $\mathtt{m}(\mathbb{t}_t, \overline{\mathbb{t}})\{\, \mathbb{b} \,\} : \mathbb{t}_r$, where $\mathbb{t}_t$ is the type value of the receiver of the method, $\overline{\mathbb{t}}$ are the type value of the arguments, $\mathbb{b}$ is the abstract behaviour of the body, and $\mathbb{t}_r$ is the type value of the returned object. The subterm $\mathbb{t}_t, \overline{\mathbb{t}}$ of the method contract is called *header*; $\mathbb{t}_r$ is called *returned type value.* We assume that names in the header occur linearly. Names in the header *bind* the names in $\mathbb{b}$ and in $\mathbb{t}_r$. The header and the returned type value, written $(\mathbb{t}_t, \overline{\mathbb{t}}) \to \mathbb{t}_r$, are called *behavioural type signature.* Names occurring in $\mathbb{b}$ or $\mathbb{t}_r$ may be *not bound* by header. These *free names* correspond to new cog creations and will be replaced by fresh cog names during the analysis. We use $\mathbb{C}$ to range over method behavioural types.

The type system uses judgments of the following form:

– $\Gamma \vdash e : \mathbb{x}$ for pure expressions e, $\Gamma \vdash f : \mathbb{t}$ or $\Gamma \vdash f : \mathbb{t}^{\checkmark}$ for future names f, and $\Gamma \vdash \mathtt{m}(\overline{\mathbb{t}}) : \mathbb{t}$ for methods.
– $\Gamma \vdash z : \mathbb{x},\ [\![\mathbb{a} \rhd \Gamma']\!]$ for expressions with side effects z, where $\mathbb{x}$ is the value, $\mathbb{a} \rhd \Gamma'$ is the corresponding behavioural type, where Γ' is the environment Γ *with possible updates* of variables and future names.
– $\Gamma \vdash s : \mathbb{b}$, in this case the updated environments Γ' are inside the behavioural type, in correspondence of every branch of its.

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

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

The *multi-hole contexts* $\mathcal{C}[\,]$ are defined by the following syntax:

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

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

The typing rules for expressions are defined in Figure 3. These rules are not standard because (size) expressions containing method's arguments are typed with the expressions themselves. This is crucial to the cost analysis in Section 5. In particular, *cog creation* is typed by rule T-New, with value $c[se]$, where c is the fresh name associated with the new cog and se is the value associated with the declared capacity. The behavioural type for the cog creation is $\nu c[se] \rhd \Gamma[c \mapsto se]$, where the newly created cog is added to Γ. In this way, it is possible to verify whether the receiver of a method invocation is within a locally created cog or not by testing whether the receiver belongs to $\mathrm{dom}(\Gamma)$ or not, respectively (*cf.* rule T-Invoke). *Object creation* (*cf.* rule T-New-Local) is typed as the cog creation, with the exception that the cog name and the capacity value are taken from the local cog and the behavioural type is empty. Rule T-Invoke types *method invocations* $e!\mathtt{m}(\overline{e})$ by using a fresh future name f that is associated to the method name, the cog name of the callee and the

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

(T-Se)
$$\Gamma \vdash se : se$$

(T-Nse)
$$\Gamma \vdash nse : -$$

(T-Method)
$$\frac{\Gamma(\mathtt{m}) = (\overline{\mathbb{t}}) \rightarrow \mathbb{t}' \qquad fv(\mathbb{t}') \setminus fv(\overline{\mathbb{t}}) \neq \varnothing \quad \text{implies} \quad \sigma(\mathbb{t}') \text{ fresh}}{\Gamma \vdash \mathtt{m}(\sigma(\overline{\mathbb{t}})) : \sigma(\mathbb{t}')}$$

(T-New)
$$\frac{\Gamma \vdash e : se \qquad c \text{ fresh}}{\Gamma \vdash \mathtt{new\ Class\ with}\ e : c[se],\ \big[\nu c[se] \rhd \Gamma[c \mapsto se]\big]}$$

(T-New-Local)
$$\frac{\Gamma \vdash \mathtt{this} : c[se]}{\Gamma \vdash \mathtt{new\ local\ Class} : c[se],\ \big[\mathtt{0} \rhd \Gamma\big]}$$

(T-Invoke-Sync)
$$\frac{\Gamma \vdash e : c[se] \quad \Gamma(\mathtt{this}) = c[se] \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{t}} \quad \Gamma \vdash \mathtt{m}(c[se], \overline{\mathbb{t}}) : \mathbb{t}'}{\Gamma \vdash e.\mathtt{m}(\overline{e}) : \mathbb{t}',\ \big[\mathtt{m}(c[se], \overline{\mathbb{t}}) \rightarrow \mathbb{t}' \rhd \Gamma\big]}$$

(T-Invoke)
$$\frac{\Gamma \vdash e : c[se] \quad (c \in \mathrm{dom}(\Gamma) \quad \text{or} \quad \Gamma(\mathtt{this}) = c[se]) \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{t}} \quad \Gamma \vdash \mathtt{m}(c[se], \overline{\mathbb{t}}) : \mathbb{t}' \quad f \text{ fresh}}{\Gamma \vdash e!\mathtt{m}(\overline{e}) : f,\ \big[\nu f \colon \mathtt{m}(c[se], \overline{\mathbb{t}}) \rightarrow \mathbb{t}' \rhd \Gamma[f \mapsto \mathbb{t}']\big]}$$

(T-Get)
$$\frac{\Gamma \vdash e : f \quad \Gamma(f) = \mathbb{t}}{\Gamma \vdash e.\mathtt{get} : \mathbb{t},\ \big[f^{\checkmark} \rhd \Gamma[f \mapsto \mathbb{t}^{\checkmark}]\big]}$$

(T-Get-Top)
$$\frac{\Gamma \vdash e : f \quad \Gamma(f) = \mathbb{t}^{\checkmark}}{\Gamma \vdash e.\mathtt{get} : \mathbb{t},\ \big[\mathtt{0} \rhd \Gamma\big]}$$

Fig. 3. Typing rules for expressions

arguments. In the updated environment, f is associated with the returned value. Next we discuss the constraints in the premise of the rule. As we discussed in Section 2, asynchronous invocations are allowed on callees located in the current cog, $\Gamma(\mathtt{this}) = c[se]$, or on a newly created object which resides in a fresh cog, $c \in \mathrm{dom}(\Gamma)$. Rule T-Get defines the *synchronization* with a method invocation that corresponds to a future f. The expression is typed with the value $\mathbb{t}$ of f in the environment and behavioural type $f^{\checkmark}$. Γ is then updated for recording that the synchronization has been already performed, thus any subsequent synchronization on the same value would not imply any waiting time (see that in rule T-Get-Top the behavioural type is $\mathtt{0}$). The *synchronous method invocation* in rule T-Invoke-Sync is directly typed with the return value $\mathbb{t}'$ of the method and with the corresponding behavioural type. The rule enforces that the cog of the callee coincides with the local one.

The typing rules for statements are presented in Figure 4. The behavioural type in rule T-Job expresses the time consumption for an object with capacity se' to perform se processing cycles: this time is given by se/se', which we observe is in general a rational number. We will return to this point in Section 5.

The typing rules for method and class declarations are shown in Figure 5.

Examples The behavioural type of the `fib` method discussed in Section 2 is

```
fib(c[x],n) {
   (n $\leq$ 1){ $0 \rhd \varnothing$ }
 + (n $\geq$ 2){
    1/x $⨾$ d[x] $⨾$ $\nu f$: fib(c[x],n-1)$\rightarrow$ - $⨾$ $\nu g$: fib(d[x],n-2)$\rightarrow$ - $⨾$
    $f^{\checkmark}$$⨾$ $g^{\checkmark}$$⨾$$0 \rhd \varnothing$ } } : -
```

(T-Assign)
$$\frac{\Gamma \vdash rhs : \mathbb{x}, \ \big[\mathbb{a} \rhd \Gamma'\big]}{\Gamma \vdash x = rhs : \mathbb{a} \rhd \Gamma'[x \mapsto \mathbb{x}]}$$

(T-Job)
$$\frac{\Gamma \vdash e : se \quad \Gamma \vdash \texttt{this} : c[se']}{\Gamma \vdash \texttt{job}(e) : se/se' \rhd \Gamma}$$

(T-Return)
$$\frac{\Gamma \vdash e : \mathbb{t} \quad \Gamma \vdash \texttt{destiny} : \mathbb{t}}{\Gamma \vdash \texttt{return}\ e : 0 \rhd \Gamma}$$

(T-Seq)
$$\frac{\begin{array}{c}\Gamma \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \\ \Gamma_i \vdash s' : \mathbb{b}'_i\end{array}}{\Gamma \vdash s\ ;\ s' : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{b}'_1] \cdots [\mathbb{a}_n \fatsemi \mathbb{b}'_n]}$$

(T-If-Nse)
$$\frac{\Gamma \vdash e : - \quad \Gamma \vdash s : \mathbb{b} \quad \Gamma \vdash s' : \mathbb{b}'}{\Gamma \vdash \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s' \,\} : \mathbb{b} + \mathbb{b}'}$$

(T-If-Se)
$$\frac{\Gamma \vdash e : se \quad \Gamma \vdash s : \mathbb{b} \quad \Gamma \vdash s' : \mathbb{b}'}{\Gamma \vdash \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s' \,\} : (se)\{\mathbb{b}\} + (\neg se)\{\mathbb{b}'\}}$$

Fig. 4. Typing rules for statements

(T-Method)
$$\frac{\begin{array}{c}\Gamma(\texttt{m}) = (\mathbb{t}_t, \overline{\mathbb{t}}) \to \mathbb{t}_r \\ \Gamma[\texttt{this} \mapsto \mathbb{t}_t][\texttt{destiny} \mapsto \mathbb{t}_r][\overline{x} \mapsto \overline{\mathbb{t}}] \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]\end{array}}{\Gamma \vdash T\ \texttt{m}\ (\overline{T\ x})\ \{\, s \,\} : \texttt{m}(\mathbb{t}_t, \overline{\mathbb{t}})\{\, \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing] \,\} : \mathbb{t}_r}$$

(T-Class)
$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \quad \Gamma[\texttt{this} \mapsto \mathrm{start}[k]][\overline{x} \mapsto \overline{\mathbb{t}}] \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\, \overline{T\ x\ ;}\ s \,\}\ \texttt{with}\ k\ : \overline{\mathbb{C}}, \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing]}$$

Fig. 5. Typing rules for declarations

5 The time analysis

The behavioural types returned by the system defined in Section 4 are used to compute upper bounds of time complexity of a `tml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [?] – and, in this section, we discuss the translation of a behavioural type program into a set of *cost equations* that are fed to the solver. 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 symbol applications (we do not define the syntax of *exp*, which may be derived by the following equations; the reader may refer to [?]), and *se* is a size expression whose variables are contained in $\overline{x}$. Basically, our translation maps method types into cost equations, where (i) method invocations are translated into function applications, and (ii) cost expressions *se* occurring in the types are left unmodified. The difficulties of the translation is that the cost equations must account for the parallelism of processes in different cogs and for sequentiality of processes in the same cog. For example, in the following code:

```
x = new Class with c; y = new Class with d;
```

```
f = x!m(); g = y!n(); u = g.get; u = f.get;
```

the invocations of `m` and `n` will run in parallel, therefore their cost will be $\mathtt{max}(t, t')$, where t is the time of executing `m` on `x` and t' is the time executing `n` on `y`. On the contrary, in the code

```
x = new local Class; y = new local Class;
f = x!m(); g = y!n(); u = g.get; u = f.get;
```

the two invocations are queued for being executed on the same cog. Therefore the time needed for executing them will be $t + t'$, where t is time needed for executing `m` on `x`, and t' is the time needed for executing `n` on `y`. To abstract away the execution order of the invocations, the execution time of *all unsynchronized* methods from the same cog are taken into account when one of these methods is synchronized with a `get`-statement. To avoid calculating the execution time of the rest of the unsynchronized methods in the same cog more than necessary, their estimated cost are ignored when they are later synchronized.

In this example, when the method invocation `y!n()` is synchronized with `g.get`, the estimated time taken is $t + t'$, which is the sum of the execution time of the two unsynchronized invocations, including the time taken for executing `m` on `x` because both `x` and `y` are residing in the same cog. Later when synchronizing the method invocation `x!m()`, the cost is considered to be *zero* because this invocation has been taken into account earlier.

The translate function. The translation of behavioural types into cost equations is carried out by the function `translate`, defined below. This function parses atoms, behavioural types or declarations of methods and classes. We will use the following auxiliary function that removes cog names from (tuples of) $\mathbb{t}$ terms:

$$\lfloor \_ \rfloor = \_ \qquad \lfloor e \rfloor = e \qquad \lfloor c[e] \rfloor = e \qquad \lfloor \mathbb{t}_1, \ldots, \mathbb{t}_n \rfloor = \lfloor \mathbb{t}_1 \rfloor, \ldots, \lfloor \mathbb{t}_n \rfloor$$

We will also use *translation environments*, ranged over by Ψ, Ψ', $\cdots$, which map future names to pairs $(e, \mathtt{m}(\overline{\mathbb{t}}))$ that records the (over-approximation of the) time when the method has been invoked and the invocation.

In the case of atoms, `translate` takes four inputs: a *translation environment* Ψ, the cog name of the carrier, an over-approximated cost e of an execution branch, and the atom $\mathbb{a}$. In this case, `translate` returns an updated translation environment and the cost. It is defined as follows.

$$
\mathtt{translate}(\Psi, c, e, \mathbb{a}) \; = \;
\begin{cases}
(\Psi, e + e') & \text{when } \mathbb{a} = e' \\
(\Psi, e) & \text{when } \mathbb{a} = \nu c[e'] \\
(\Psi, e + \mathtt{m}(\lfloor\overline{\mathbb{t}}\rfloor)) & \text{when } \mathbb{a} = \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}' \\
(\Psi[f \mapsto (e, \mathtt{m}(\overline{\mathbb{t}}))],\, e) & \text{when } \mathbb{a} = (\nu f\colon \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}') \\
(\Psi \setminus F, e + e_1))) & \text{when } \mathbb{a} = f^{\checkmark} \quad \text{and} \quad \Psi(f) = (e_f, \mathtt{m}_f(c[e'], \overline{\mathbb{t}_f})) \\
 & \text{let } F = \{\, g \mid \Psi(g) = (e_g,\, \mathtt{m}_g(c[e'], \overline{\mathbb{t}_g}))\,\} \text{ then} \\
 & \text{and } e_1 = \sum \{\, \mathtt{m}_g(\lfloor\overline{\mathbb{t}'_g}\rfloor) \mid (e_g, \mathtt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F)\,\} \\
(\Psi \setminus F, \mathit{max}(e, e_1 + e_2)) & \text{when } \mathbb{a} = f^{\checkmark} \text{ and } \Psi(f) = (e_f, \mathtt{m}_f(c'[e'], \overline{\mathbb{t}_f})) \text{ and } c \neq c' \\
 & \text{let } F = \{\, g \mid \Psi(g) = (e_g,\, \mathtt{m}_g(c'[e'], \overline{\mathbb{t}_g}))\,\} \text{ then} \\
 & e_1 = \sum \{\, \mathtt{m}_g(\lfloor\overline{\mathbb{t}'_g}\rfloor) \mid (e_g, \mathtt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F)\,\} \\
 & \text{and } e_2 = \mathit{max}\{\, e_g \mid (e_g, \mathtt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F)\,\} \\
(\Psi, e) & \text{when } \mathbb{a} = f^{\checkmark} \text{ and } f \notin \mathrm{dom}(\Psi)
\end{cases}
$$

The interesting case of `translate` is when the atom is $f^{\checkmark}$. There are three cases:

1. The synchronization is with a method whose callee is an object of the same cog. In this case its cost must be *added*. However, it is not possible to know when the method will be actually scheduled. Therefore, we sum the costs of all the methods running on the same cog (worst case) – the set F in the formula – and we remove them from the translation environment.
2. The synchronization is with a method whose callee is an object on a different cog c'. In this case we use the cost that we stored in $\Psi(f)$. Let $\Psi(f) = (e_f, \mathtt{m}_f(c'[e'], \overline{\mathbb{t}_f}))$, then e_f represents the time of the invocation. The cost of the invocation is therefore $e_f + \mathtt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor)$. Since the invocation is *in parallel* with the thread of the cog c, the overall cost is $\mathit{max}(e, e_f + \mathtt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor))$. As in case 1, we consider the worst scheduler choice on c'. Instead of taking $e_f + \mathtt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor)$, we compute the cost of all the methods running on c' – the set F in the formula – and we remove them from the translation environment.
3. The future does not belong to Ψ. That is the cost of the invocation which has been already computed. In this case, the value e does not change.

In the case of behavioural types, `translate` takes as input a translation environment, the cog name of the carrier, an over-approximated cost of the current execution branch $(e_1)e_2$, where e_1 indicates the conditions corresponding to the branch, and the behavioural type $\mathbb{a}$.

$$
\mathtt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}) \; = \;
\begin{cases}
\{\, (\Psi', (e_1)e'_2)\,\} & \text{when } \mathbb{b} = \mathbb{a} \rhd \Gamma \quad \text{and} \quad \mathtt{translate}(\Psi, c, e_2, \mathbb{a}) = (\Psi', e'_2) \\
C & \text{when } \mathbb{b} = \mathbb{a} \fatsemi \mathbb{b}' \quad \text{and} \quad \mathtt{translate}(\Psi, c, e_2, \mathbb{a}) = (\Psi', e'_2) \\
 & \text{and } \mathtt{translate}(\Psi', c, (e_1)e'_2, \mathbb{b}') = C \\
C \cup C' & \text{when } \mathbb{b} = \mathbb{b}_1 + \mathbb{b}_2 \quad \text{and} \quad \mathtt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}_1) = C \\
 & \text{and} \quad \mathtt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}_2) = C' \\
C & \text{when } \mathbb{b} = (e)\{\, \mathbb{b}'\,\} \quad \text{and} \quad \mathtt{translate}(\Psi, c, (e_1 \wedge e)e_2, \mathbb{b}') = C
\end{cases}
$$

The translation of the behavioural types of a method is given below. Let $\mathrm{dom}(\Psi) = \{\, f_1, \cdots, f_n\,\}$. Then we define $\Psi^{\checkmark} \overset{def}{=} {f_1}^{\checkmark} \fatsemi \cdots \fatsemi {f_n}^{\checkmark}$.

$$\texttt{translate}(\texttt{m}(c[e],\overline{\mathbb{t}})\{\,\mathbb{b}\,\}:\mathbb{t}) \;=\; \left[\begin{array}{ll} \texttt{m}(e,\overline{e}) = e'_1 + e''_1 & [e_1] \\ \vdots & \\ \texttt{m}(e,\overline{e}) = e'_n + e''_n & [e_n] \end{array}\right.$$

where $\texttt{translate}(\varnothing, c, \texttt{0}, \mathbb{b}) = \{\,\Psi_i, (e_i)e'_i \mid 1 \le i \le n\,\}$, and $\overline{e} = \lfloor\overline{\mathbb{t}}\rfloor$, and $e''_i = \texttt{translate}(\Psi_i, c, \texttt{0}, \Psi_i^{\checkmark} \triangleright \varnothing)$. In addition, $[e_i]$ are the conditions for branching the possible execution paths of method $\texttt{m}(e,\overline{e})$, and $e'_i + e''_i$ is the over-approximation of the cost for each path. In particular, e'_i corresponds to the cost of the synchronized operations in each path (e.g., `job`s and `get`s), while e''_i corresponds to the cost of the asynchronous method invocations triggered by the method, but not synchronized within the method body.

Examples We show the translation of the behavioural type of fibonacci presented in Section 4. Let $\mathbb{b} = (se)\{\texttt{0} \triangleright \varnothing\} + (\neg se)\{\mathbb{b}'\}$, where $se = (\text{n} \le 1)$ and $\mathbb{b}' = 1/e \,⨾\, \nu f\colon \texttt{fib}(c[e], n-1) \to \texttt{-} \,⨾\, \nu g\colon \texttt{fib}(c'[e], n-2) \to \texttt{-} \,⨾\, f^{\checkmark} \,⨾\, g^{\checkmark} \,⨾\, \texttt{0} \triangleright \varnothing\}$. Let also $\Psi = \Psi_1 \cup \Psi_2$, where $\Psi_1 = [f \mapsto (1/e, \texttt{fib}(e, n-1))]$ and $\Psi_2 = [g \mapsto (1/e, \texttt{fib}(e, n-2))]$.

The following equations summarize the translation of the behavioural type of the fibonacci method.

$$\begin{aligned}
&\texttt{translate}(\varnothing, c, 0, \mathbb{b}) \\
&\quad = \texttt{translate}(\varnothing, c, 0, (se)\ \{\,\texttt{0} \triangleright \varnothing\,\}) \ \cup\ \texttt{translate}(\varnothing, c, 0, (\neg se)\ \{\,\mathbb{b}'\,\}) \\
&\quad = \texttt{translate}(\varnothing, c, (se)0, \{\,\texttt{0} \triangleright \varnothing\,\}) \ \cup\ \texttt{translate}(\varnothing, c, (\neg se)0, \{\,1/e \ ⨾\ \ldots\})\\
&\quad = \{\,(se)0\,\} \ \cup\ \texttt{translate}(\varnothing, c, (\neg se)(1/e), \{\,\nu f\colon \texttt{fib}(c[e], n-1) \to \texttt{-}\ ⨾ \ldots\}) \\
&\quad = \{\,(se)0\,\} \ \cup\ \texttt{translate}(\Psi_1, c, (\neg se)(1/e), \{\,\nu g\colon \texttt{fib}(c'[e], n-2) \to \texttt{-}\ ⨾ \ldots\}) \\
&\quad = \{\,(se)0\,\} \ \cup\ \texttt{translate}(\Psi, c, (\neg se)(1/e), \{\,f^{\checkmark} ⨾ g^{\checkmark} ⨾ \ldots\}) \\
&\quad = \{\,(se)0\,\} \ \cup\ \texttt{translate}(\Psi_2, c, (\neg se)(1/e + \texttt{fib}(e, n-1)), \{\,g^{\checkmark} ⨾ \ldots\}) \\
&\quad = \{\,(se)0\,\} \ \cup\ \texttt{translate}(\varnothing, c, (\neg se)(1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2))), \{\,\texttt{0} \triangleright \varnothing\,\}) \\
&\quad = \{\,(se)0\,\} \ \cup\ \{\,(\neg se)(1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2)))\,\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{translate}(\varnothing, c, 0, 0) \;=\; (\varnothing, 0) \\
&\texttt{translate}(\varnothing, c, 0, 1/e) \;=\; (\varnothing, 1/e) \\
&\texttt{translate}(\varnothing, c, 1/e, \nu f\colon \texttt{fib}(c[e], n-1) \to \texttt{-}\ ) \;=\; (\Psi_1, 1/e) \\
&\texttt{translate}(\Psi_1, c, 1/e, \nu g\colon \texttt{fib}(c'[e], n-2) \to \texttt{-}\ ) \;=\; (\Psi, 1/e) \\
&\texttt{translate}(\Psi, c, 1/e, f^{\checkmark}\ ) \;=\; (\Psi_2, 1/e + \texttt{fib}(e, n-1)) \\
&\texttt{translate}(\Psi_2, c, 1/e + \texttt{fib}(e, n-1), g^{\checkmark}\ ) \;=\; (\varnothing, 1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{translate}(\texttt{fib}\ (c[e], n)\{\,\mathbb{b}\,\} : \texttt{-}) \;=\; \\
&\begin{cases} \texttt{fib}(e, n) = 0 & [n\ \le 1] \\ \texttt{fib}(e, n) = 1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2)) & [n\ \ge 2] \end{cases}
\end{aligned}$$

Remark 1. Rational numbers are produced by the rule T-Job of our type system. In particular behavioural types may manifest terms se/se' where se gives the processing cycles defined by the `job` operation and se' specifies the number of processing cycles per unit of time the corresponding cog is able to handle. Unfortunately, our backend solver – `CoFloCo` – cannot handle rationals se/se'

where se' is a variable. This is the case, for instance, of our fibonacci example, where the cost of each iteration is `1/x`, where `x` is a parameter. In order to analyse this example, we need to determine *a priori* the capacity to be a constant – say `2` –, obtaining the following input for the solver:

```
eq(f(E,N),0,[],[-N>=1,2*E=1]).
eq(f(E,N),nat(E),[f(E,N-1)],[N>=2,2*E=1]).
eq(f(E,N),nat(E),[f(E,N-2)],[N>=2,2*E=1]).
```

Then the solver gives `nat(N-1)*(1/2)` as the upper bound. It is worth to notice that fixing the fibonacci method is easy because the capacity does not change during the evaluation of the method. This is not always the case, as in the following alternative definition of fibonacci:

```
Int fib_alt(Int n) {
    if (n<=1) { return 1; }
    else { Fut<Int> f; Class z; Int m1; Int m2;
           job(1);
           z = new Class with (this.capacity*2) ;
           f = this!fib_alt(n-1); g = z!fib_alt(n-2);
           m1 = f.get; m2 = g.get;
           return m1+m2; } }
```

In this case, the recursive invocation `z!fib_alt(n-2)` is performed on a cog with twice the capacity of the current one and `CoFloCo` is not able to handle it. It is worth to observe that this is a problem of the solver, which is otherwise very powerful for most of the examples. Our behavioural types carry enough information for dealing with more complex examples, so we will consider alternative solvers or combination of them for dealing with examples like `fib_alt`.

6 Properties

In order to prove the correctness of our system, we need to show that (*i*) the behavioural type system is correct, and (*ii*) the computation time returned by the solver is an upper bound of the actual cost of the computation.

The correctness of the type system in Section 4 is demonstrated by means of a subject reduction theorem expressing that if a runtime configuration cn is well typed and $cn \to cn'$ then cn' is well-typed as well, and the computation time of cn is larger or equal to that of cn'. In order to formalize this theorem we extend the typing to configurations and we also use extended behavioural types $\mathbb{k}$ with the following syntax

$$\mathbb{k} ::= \quad \mathbb{b} \;\mid\; [\mathbb{b}]_f^c \;\mid\; \mathbb{k} \parallel \mathbb{k} \qquad\qquad \text{runtime behavioural type}$$

The type $[\mathbb{b}]_f^c$ expresses the behaviour of an asynchronous method bound to the future f and running in the cog c; the type $\mathbb{k} \parallel \mathbb{k}'$ expresses the parallel execution of methods in $\mathbb{k}$ and in $\mathbb{k}'$.

We then define a relation $\trianglerighteq_t$ between runtime behavioural types that relates types. The definition is algebraic, and $\mathbb{k} \trianglerighteq_t \mathbb{k}'$ is intended to mean that the computational time of $\mathbb{k}$ is at least that of $\mathbb{k}'+t$ (or conversely the computational time of $\mathbb{k}'$ is at most that of $\mathbb{k}-t$). This is actually the purpose of our theorems.

Theorem 1 (Subject Reduction). *Let cn be a configuration of a* `tml` *program and let* $\Bbbk$ *be its behavioural type. If cn is not strongly t-stable and* $cn \rightarrow cn'$ *then there exists* $\Bbbk'$ *typing* cn' *such that* $\Bbbk \trianglerighteq_0 \Bbbk'$. *If cn is strongly t-stable and* $cn \rightarrow cn'$ *then there exists* $\Bbbk'$ *typing* cn' *such that* $\Bbbk \trianglerighteq_t \Bbbk'$.

The proof of is a standard case analysis on the last reduction rule applied. The second part of the proof requires an extension of the `translate` function to run-time behavioural types. We therefore define a cost of the equations $\mathcal{E}_{\Bbbk}$ returned by $\texttt{translate}(\Bbbk)$ – noted $\texttt{cost}(\mathcal{E}_{\Bbbk})$ – by unfolding the equational definitions.

Theorem 2 (Correctness). *If* $\Bbbk \trianglerighteq_t \Bbbk'$, *then* $\texttt{cost}(\mathcal{E}_{\Bbbk}) \ \geq \ \texttt{cost}(\mathcal{E}_{\Bbbk'}) + t$.

As a byproduct of Theorems 1 and 2, we obtain the correctness of our technique, modulo the correctness of the solver.

7 Related work

In contrast to the static time analysis for sequential executions proposed in [**?**], the paper proposes an approach to analyse time complexity for concurrent programs. Instead of using a Hoare-style proof system to reason about end-user deadlines, we estimate the execution time of a concurrent program by deriving the time-consuming behaviour with a type-and-effect system.

Static time analysis approaches for concurrent programs can be divided into two main categories: those based on type-and-effect systems and those based on abstract interpretation – see references in [**?**]. Type-and-effect systems (i) collect constraints on type and resource variables and (ii) solve these constraints. The difference with respect to our approach is that we do not perform the analysis during the type inference. We use the type system for deriving behavioural types of methods and, in a second phase, we use them to run a (non compositional) analysis that returns cost upper bounds. This dichotomy allows us to be more precise, avoiding unification of variables that are performed during the type derivation. In addition, we notice that the techniques in the literature are devised for programs where parallel modules of sequential code are running. The concurrency is not part of the language, but used for parallelising the execution.

Abstract interpretation techniques have been proposed addressing domains carrying quantitative information, such as resource consumption. One of the main advantages of abstract interpretation is the fact that many practically useful optimization techniques have been developed for it. Consequently, several well-developed automatic solvers for cost analysis already exist. These techniques either use finite domains or use expedients (widening or narrowing functions) to guarantee the termination of the fix-point generation. For this reason, solvers often return inaccurate answers when fed with systems that are finite but not statically bounded. For instance, an abstract interpretation technique that is very close to our contribution is [**?**]. The analysis of this paper targets a language with the same concurrency model as ours, and the backend solver for our analysis, `CoFloCo`, is a slightly modified version of the solver used by [**?**]. However the two techniques differ profoundly in the resulting cost equations and in the way they are produced. Our technique computes the cost by means of a type system, therefore every method has an associated type, which is parametric with respect to the arguments. Then these types are translated into a bunch of cost equations that may be *composed* with those of other methods. So our approach supports a technique similar to *separate compilation*, and is able to deal with systems that create statically an unbounded but finite number of nodes. On the contrary, the technique in [?] is not compositional because it takes the whole program and computes the parts that may run in parallel. Then the cost equations are generated accordingly. This has the advantage that their technique does not have any restriction on invocations on arguments of methods that are (currently) present in our one.

We finally observe that our behavioural types may play a relevant role in a cloud computing setting because they may be considered as abstract descriptions of a method suited for SLA compliance.

8 Conclusions

This article presents a technique for computing the time of concurrent object-oriented programs by using behavioural types. The programming language we have studied features an explicit cost annotation operation that define the number of machine cycles required before executing the continuation. The actual computation activities of the program are abstracted by `job`-statements, which are the unique operations that consume time. The computational cost is then measured by introducing the notion of (strong) *t-stability* (*cf.* Definition 1), which represents the ticking of time and expresses that up to t time steps no control activities are possible. A Subject Reduction theorem (Theorem 1), then, relates this stability property to the derived types by stating that the consumption of t time steps by `job` statements is properly reflected in the type system. Finally, Theorem 2 states that the solution of the cost equations obtained by translation of the types provides an upper bound of the execution times provided by the type system and thus, by Theorem 1, of the actual computational cost.

Our behavioural types are translated into so-called cost equations that are fed to a solver that is already available in the literature – the `CoFloCo` solver [?]. As discussed in Remark 1, `CoFloCo` cannot handle rational numbers with variables at the denominator. In our system, this happens very often. In fact, the number pc of processing cycles needed for the computation of a `job`(pc) is divided by the speed s of the machine running it. This gives the cost in terms of time of the `job`(pc) statement. When the capacity is not a constant, but depends on the value of some parameter and changes over time, then we get the untreatable rational expression. It is worth to observe that this is a problem of the solver (otherwise very powerful for most of the examples), while our behavioural types carry enough information for computing the cost also in these cases. We plan to consider alternative solvers or a combination of them for dealing with complex examples.

Our current technique does not address the full language. In particular we are still not able to compute costs of methods that contain invocations to arguments which do not live in the same machine (which is formalized by the notion of cog in our language). In fact, in this case it is not possible to estimate the cost without any indication of the state of the remote machine. A possible solution to this issue is to deliver costs of methods that are parametric with respect to the state of remote machines passed as argument. We will investigate this solution in future work.

In this paper, the cost of a method also includes the cost of the asynchronous invocations in its body that have not been synchronized. A more refined analysis, combined with the resource analysis of [**?**], might consider the cost of each machine, instead of the overall cost. That is, one should count the cost of a method *per* machine rather than in a cumulative way. While these values are identical when the invocations are always synchronized, this is not the case for unsynchronized invocation and a disaggregated analysis might return better estimations of virtual machine usage.