Chapter 2
A Type System for Components

In this chapter we present the type system for the component model. We first give a thorough explanation of the types we adopt and how the type system achieves tracking of cog membership. Then, we introduce the subtyping relation; we present the auxiliary functions and predicates that the type system relies on, and we conclude with the typing rules.

2.1 Typing Features

In this section we give the intuition behind the types and the records used in the typing rules, the latter being a new concept not adopted either in ABS or in its component extension [77]. We explain also the meaning of the method signature and how the type system addresses the problem of consistent rebindings and consistent synchronous method calls.

Cog Names The goal of our type system is to statically check if rebindings and synchronous method calls are performed locally to a cog. Since cogs and objects are entities created at runtime, we cannot know statically their identity. The interesting, and also difficult part, in designing the type systems is how to statically track cogs identity and hence membership to a cog. We address this issue by using a *linear* type system on names of cogs, which range over $\mathtt{G}, \mathtt{G}', \mathtt{G}''$, in a way that abstracts the runtime identity of cogs. The type system associates to every cog creation a unique cog name, which makes it possible to check if two objects are in the same cog or not.

Precisely, we associate objects to their cogs using records $\mathtt{r}$, having the form $\mathtt{G}[\overline{f : T}]$, where $\mathtt{G}$ denotes the cog in which the object is located and $[\overline{f : T}]$ maps any object's fields in $\overline{f}$ to its type in $\overline{T}$. In fact, in order to correctly track cog membership of each expression, we also need to keep information about the cog of the object's fields in a record. This is needed, for instance, when an object stored in

O. Dardha, *Type Systems for Distributed Programs: Components and Sessions*, Atlantis Studies in Computing 7,
DOI 10.2991/978-94-6239-204-5_2

a field is accessed within the method body and then returned by the method; in this case one needs a way to bind the cog of the accessed field to the cog of the returned value.

Cog Sets In order to deal with linearity of cogs created, and to keep track of them after their creation, our type system, besides the standard typing context Γ (formally defined in the next section) uses a set of cogs, ranged over by $\mathcal{G}$, $\mathcal{G}'$, $\mathcal{G}''$, that keeps track of the cogs created so far and uses the operator $\uplus$ to deal with the disjoint union of sets, namely $\mathcal{G} \uplus \mathcal{G}'$, where the empty set acts as the neutral element, namely $\mathcal{G} \uplus \emptyset = \emptyset \uplus \mathcal{G} = \mathcal{G}$. We will discuss the details in Sect. 2.4.

Method Signature Let us now explain the method signature $(\mathcal{G}, \mathbb{r})$ used to annotate a method header. The record $\mathbb{r}$ is used as the record of the object **this** during the typing of the method, i.e., $\mathbb{r}$ is the binder for the cog of the object **this** in the scope of the method body, as we will see in the typing rules in the following. The set of cog names $\mathcal{G}$ is used to keep track of the fresh cogs that the method creates. In particular, when we deal with recursive method calls, the set $\mathcal{G}$ gathers the fresh cogs of every call, which is then returned to the main execution. Moreover, when it is not necessary to keep track of cog information about an object, because the object is not going to take part in any synchronous method call or any rebind operation, it is possible to associate to this object the *unknown* record $\perp$. This special record does not keep any information about the cog where the object or its fields are located, and it is to be considered different from any other cog, thus to ensure the soundness of our type system. Finally, notice that data types also may contain records; for instance, a list of objects is typed with $\mathrm{List}\langle T\rangle$ where T is the type of the objects in the list and it may include the records of the objects.

2.2 Subtyping Relation

There are two forms of subtyping: *structural* and *nominal* subtyping. In a language where subtyping is nominal, A is a subtype of B if and only if it is declared to be so, meaning if class (or interface) A extends (or implements) class (or interface) B; these relations must be defined by the programmer and are based on the names of classes and interfaces declared. In the latter, subtyping relation is established by analysing the structure of a class, i.e., its fields and methods: class (or interface) A is a subtype of class (or interface) B if and only if the fields and methods of A are a superset of the fields and methods of B, and their types in A are subtypes of their types in B. (Featherweight) Java uses nominal subtyping, languages like [44, 52, 81, 92] use structural subtyping. In [33] the authors integrate both nominal and structural subtyping.

The subtyping relation $\leq$ for our language is given in Fig. 2.1; we adopt both nominal and structural subtyping. Rule (S-DATA) states that data types are covariant in their type parameters. Rule (S-TYPE) states that annotating classes and interfaces

$$\text{(S-D{\scriptsize ATA})}\quad \frac{\forall i \quad T_i \le T'_i}{D\langle \overline{T} \rangle \le D\langle \overline{T}' \rangle} \qquad\qquad \text{(S-T{\scriptsize YPE})}\quad \frac{L \le L'}{(L, \mathtt{r}) \le (L', \mathtt{r})}$$

$$\text{(S-F{\scriptsize IELDS})}\quad \frac{f \notin ports(L)}{(L, \mathsf{G}[f : T; \overline{f : T}]) \le (L, \mathsf{G}[\overline{f : T}])} \qquad \text{(S-P{\scriptsize ORTS})}\quad \frac{f \in ports(L)}{(L, \mathsf{G}[\overline{f : T}]) \le (L, \mathsf{G}[f : T; \overline{f : T}])}$$

$$\text{(S-C{\scriptsize LASS})}\quad \frac{\textbf{class}\ \mathtt{C}[(\overline{T\ x})]\ \textbf{implements}\ \overline{\mathtt{I}}\ \{\ \overline{Fl}\ \overline{M}\ \} \quad \mathtt{I}_i \in \overline{\mathtt{I}}}{\mathtt{C} \le \mathtt{I}_i} \qquad \text{(S-R{\scriptsize EFL})}\quad \frac{}{T \le T}$$

$$\text{(S-I{\scriptsize NTERFACE})}\quad \frac{\textbf{interface}\ \mathtt{I}\ \textbf{extends}\ \overline{\mathtt{I}}\ \{\ \overline{\textbf{port}\ T\ x;}\ \overline{S}\ \} \quad \mathtt{I}_i \in \overline{\mathtt{I}}}{\mathtt{I} \le \mathtt{I}_i} \qquad \text{(S-T{\scriptsize RANS})}\quad \frac{T \le T' \quad T' \le T''}{T \le T''}$$

Fig. 2.1 Subtyping relation

with records does not change the subtyping order. Rules (S-FIELDS) and (S-PORTS) use structural subtyping on records. Fields, like methods, are what the object provides, hence it is sound to forget about the existence of a field in an object. This is why the rule (S-FIELDS) allows to remove fields from records. Ports on the other hand, model the dependencies the objects have on their environment, hence it is sound to consider that an object may have more dependencies than it actually has during its execution. This is why the rule (S-PORTS) allows to add ports to records. So, in case of fields, one object can be substituted by another one if the latter has at least the same fields; on the contrary, in case of ports, one object can be substituted by another one if the latter has at most the same ports. Notice that in the standard object-oriented setting this rule would not be sound, since trying to access a non-existing attribute would lead to a null pointer exception. Therefore, to support our vision of port behaviour, we add a (REBIND-NONE) reduction rule to the component calculus semantics which simply permits the rebind to succeed without modifications if the port is not available. Rules (S-CLASS) and (S-INTERFACE) use nominal subtyping and state that a class `C` (respectively, an interface `I`) is a subtype of an interface $\mathtt{I}_i$ that it implements (respectively, extends). Rules (S-REFL) and (S-TRANS) are standard and state that our subtyping relation is a preorder.

2.3 Functions and Predicates

In this section we define the auxiliary functions and predicates that are used in the typing rules. We start with the lookup functions *params, ports, fields, ptype, mtype, heads* shown in Fig. 2.2. These functions are similar and are inspired by the corresponding ones in Featherweight Java [61]. For readability reasons, the lookup

$$\frac{\textbf{class}\ \mathsf{C}\ (\overline{T\ x})\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl};\ \overline{M}\ \}}{params(\mathsf{C}) = \overline{T\ x}}$$

$$\frac{\textbf{class}\ \mathsf{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{ports(\mathsf{C}) = \overline{T\ x}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{ports(\mathtt{I}) = \overline{T\ x}}$$

$$\frac{\textbf{class}\ \mathsf{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{fields(C) = \overline{T\ x}; \overline{T'\ x'}}$$

$$\frac{\textbf{class}\ \mathsf{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{ptype(p, \mathsf{C}) = \overline{T}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{ptype(p, \mathtt{I}) = \overline{T}}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathsf{C}\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \ \overline{M}\ \}\\ [\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x})\{\ s\ \} \in \overline{M}\end{array}}{mtype(\mathtt{m}, \mathsf{C}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T}$$

$$\frac{\begin{array}{c}\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}\\ [\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x}) \in \overline{S}\end{array}}{mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T}$$

$$\frac{\textbf{class}\ \mathsf{C}\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\,\overline{M}\ \} \qquad \overline{M} = \overline{S\ \{s\}}}{heads(\mathsf{C}) = \overline{S}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{heads(\mathtt{I}) = \overline{S}}$$

Fig. 2.2 Lookup functions

functions are written in italics, whether the auxiliary functions and predicates are not. Function *params* returns the sequence of typed parameters of a class. Function *ports* returns the sequence of typed ports. Instead, function *fields* returns all the fields of the class it is defined on, namely the inner state and the ports too. Functions *ptype* and *mtype* return the declared type of respectively the port and the method they are applied to. Function *heads* returns the headers of the declared methods.

$$\text{tmatch}(T, T) = id \qquad \text{tmatch}(\mathbb{r}, \mathbb{r}) = id \qquad \text{tmatch}(\mathtt{V}, T) \triangleq [\mathtt{V} \mapsto T]$$

$$\frac{\forall i \quad \text{tmatch}(T_i, T'_i) = \sigma_i \qquad \forall i, j \quad \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)}}{\text{tmatch}(D\langle\overline{T}\rangle, D\langle\overline{T'}\rangle) \triangleq \biguplus_i \sigma_i}$$

$$\frac{\text{tmatch}(\mathbb{r}, \mathbb{r}') = \sigma}{\text{tmatch}((\mathtt{I}, \mathbb{r}), (\mathtt{I}, \mathbb{r}')) \triangleq \sigma}$$

$$\frac{\forall i \quad \text{tmatch}(T_i, T'_i) = \sigma_i \qquad \forall i, j \quad \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)} \qquad \sigma(\mathtt{G}) \in \{\mathtt{G}, \mathtt{G}'\}}{\text{tmatch}(\mathtt{G}[\overline{f : T}], \mathtt{G}'[\overline{f : T'}]) \triangleq [\mathtt{G} \mapsto \mathtt{G}'] \biguplus_i \sigma_i}$$

$$\text{pmatch}(\_, T) \triangleq \emptyset \qquad \text{pmatch}(x, T) \triangleq \emptyset; x : T \qquad \text{pmatch}(\textbf{null}, (\mathtt{I}, \mathbb{r})) \triangleq \emptyset$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{Co}) = \overline{T} \to T' \\ \text{tmatch}(T', T'') = \sigma \qquad \forall i \quad \text{pmatch}(p_i, \sigma(T_i)) = \Gamma_i\end{array}}{\text{pmatch}(\mathsf{Co}(\overline{p}), T'') \triangleq \biguplus_i \Gamma_i}$$

$$\frac{\begin{array}{c}\mathsf{C} \leq \mathtt{I} \qquad \mathsf{dom}(\sigma') \cap \mathsf{dom}(\sigma) = \emptyset \\ \mathit{fields}(\mathsf{C}) = \overline{(\mathtt{I}, \mathbb{r})\ f}; \overline{\mathtt{D}(\ldots)\ f'}\end{array}}{(\mathtt{I}, \mathtt{G}[\overline{f : \sigma \circ \sigma'(\mathtt{I}, \mathbb{r})}]) \in \text{crec}(\mathtt{G}, \mathsf{C}, \sigma)}$$

$$\frac{\text{equals}(\mathtt{G}, \mathtt{G}')}{\text{coloc}(\mathtt{G}[\ldots], (\mathsf{C}, \mathtt{G}'[\ldots]))}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathsf{C}) \subseteq \mathit{ports}(\mathtt{I}) \text{ and } \forall p \in \mathit{ports}(\mathsf{C}).\ \mathit{ptype}(p, \mathtt{I}) = \mathit{ptype}(p, \mathsf{C}) \\ \mathit{heads}(\mathtt{I}) \subseteq \mathit{heads}(\mathsf{C}) \text{ and } \forall \mathtt{m} \in \mathtt{I}.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathsf{C})\end{array}}{\text{implements}(\mathsf{C}, \mathtt{I})}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathtt{I}) \subseteq \mathit{ports}(\mathtt{I}') \text{ and } \forall p \in \mathit{ports}(\mathtt{I}).\ \mathit{ptype}(p, \mathtt{I}') = \mathit{ptype}(p, \mathtt{I}) \\ \mathit{heads}(\mathtt{I}') \subseteq \mathit{heads}(\mathtt{I}) \text{ and } \forall \mathtt{m} \in \mathtt{I}'.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathtt{I}')\end{array}}{\text{extends}(\mathtt{I}, \mathtt{I}')}$$

Fig. 2.3 Auxiliary functions and predicates

Except function $fields$ which is defined only on classes, the rest of the lookup functions is defined on both classes and interfaces.

The auxiliary functions and predicates are shown in Fig. 2.3. Function tmatch returns a substitution σ of the formal parameters to the actual ones. It is defined both on types and on records. The matching of a type T to itself, or of a record $\mathbb{r}$ to itself, returns the identity substitution id; the matching of a type variable $\mathtt{V}$ to a type T returns a substitution of $\mathtt{V}$ to T; the matching of data type D parametrized on formal types $\overline{T}$ and on actual types $\overline{T'}$ returns the union of substitutions that correspond to the matching of each type T_i with T'_i, in such a way that substitutions coincide when applied to the same formal types, the latter being expressed by $\forall i, j\ \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)}$; the matching of records follows the same idea as that of data types. Finally, tmatch applied on types $(\mathtt{I}, \mathbb{r})$, $(\mathtt{I}, \mathbb{r}')$ returns the same substitution obtained by matching $\mathbb{r}$ with $\mathbb{r}'$. Function pmatch, performs matchings on patterns and types by returning a typing context Γ. In particular, pmatch returns an empty set when the pattern is _ or **null**, or $x : T$ when applied on a variable x and a type T. Otherwise, if applied to a constructor expression $\mathtt{Co}(\overline{p})$ and a type T'' it returns the union of typing contexts corresponding to patterns in $\overline{p}$. The pair $(\mathtt{I}, \mathtt{G}[\sigma \uplus \sigma'(\overline{f : (\mathtt{I}, \mathbb{r})})])$ is a member of crec$(\mathtt{G}, \mathtt{C}, \sigma)$ if class $\mathtt{C}$ implements interface $\mathtt{I}$ and σ' and σ are substitutions defined on disjoint sets of names. Predicate coloc states the equality of two cog names. Predicates implements and extends check when a class implements an interface and an interface extends another one. A class $\mathtt{C}$ implements an interface $\mathtt{I}$ if the ports of $\mathtt{C}$ are at *most* the ones of $\mathtt{I}$. Instead, for methods, $\mathtt{C}$ may define at *least* the methods declared in $\mathtt{I}$ having the same signature. The extends predicate states when an interface $\mathtt{I}$ properly extends another interface $\mathtt{I}'$ and is defined similarly to the implements predicate.

2.4 Typing Rules

A *typing context* Γ is a partial function and assigns types T to variables, a pair $(\mathtt{C}, \mathbb{r})$ to **this**, and arrow types $\overline{T} \to T'$ to function symbols like $\mathtt{Co}$ or $\mathtt{fun}$, namely:

$$\Gamma ::= \emptyset \mid x : T, \Gamma \mid \mathbf{this} : (\mathtt{C}, \mathbb{r}), \Gamma \mid \mathtt{Co} : \overline{T} \to T', \Gamma \mid \mathtt{fun} : \overline{T} \to T', \Gamma$$

As usual $\mathsf{dom}(\Gamma)$ denotes the domain of the typing context Γ. We define the *composition* of typing contexts, $\Gamma \circ \Gamma'$, as follows: $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ if $x \in \mathsf{dom}(\Gamma')$, and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise. We say that a typing context Γ' *extends* a typing context Γ, denoted with $\Gamma \subseteq \Gamma'$ if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathsf{dom}(\Gamma)$. Typing judgements have the following forms, where a cogset $\mathcal{G}$ indicates the set of new cogs created by the term being typed. $\Gamma \vdash g : \mathtt{Bool}$ for guards; $\Gamma \vdash e : T$ for pure expressions; $\Gamma, \mathcal{G} \vdash z : T$ for expressions with side effects; $\Gamma, \mathcal{G} \vdash s$ for statements; $\Gamma \vdash M$ for method declarations; $\Gamma \vdash C$ for class declarations and $\Gamma \vdash I$ for interface declarations.

Pure Expressions The typing rules for pure expressions are given in Fig. 2.4. Rule (T-VAR/FIELD) states that a variable is of type the one assumed in the typing context. Rule (T-FIELDR) assigns to x a type T and a record $\mathbb{r}$ fetched from the type of **this**. Rule (T-FIELDBOT) assigns $(T, \bot)$ to x, since x is not part of the record for **this** but is a field of $\mathtt{C}$. Rule (T-NULL) states that the value **null** is of type any interface $\mathtt{I}$ declared in the CT (class table) and any record $\mathbb{r}$. Rule (T-WILD) states that the wildcard _ is

$$\text{(T-VAR/FIELD)}\quad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-FIELDR)}\quad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma(\mathbf{this}) = (\mathsf{C}, \mathsf{G}[x : (T, \mathbb{r}), \ldots])}{\Gamma \vdash x : (T, \mathbb{r})}$$

$$\text{(T-FIELDBOT)}\quad \frac{x \notin \mathsf{dom}(\Gamma) \qquad T\ x \in \mathit{fields}(\mathsf{C}) \qquad \Gamma(\mathbf{this}) = (\mathsf{C}, \mathsf{G}[\overline{x : T}]) \qquad x \notin \overline{x}}{\Gamma \vdash x : (T, \bot)} \qquad \text{(T-NULL)}\quad \frac{\mathbf{interface}\ \mathtt{I}\ [\cdots]\ \{\ \cdots\ \} \in CT}{\Gamma \vdash \mathbf{null} : (\mathtt{I}, \mathbb{r})} \qquad \text{(T-WILD)}\quad \frac{}{\Gamma \vdash \_ : T}$$

$$\text{(T-CONSEXP)}\quad \frac{\Gamma(\mathsf{Co}) = \overline{T} \rightarrow T' \qquad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad \Gamma \vdash \overline{e} : \overline{T'}}{\Gamma \vdash \mathsf{Co}(\overline{e}) : \sigma(T')} \qquad \text{(T-FUNEXP)}\quad \frac{\Gamma(\mathtt{fun}) = \overline{T} \rightarrow T' \qquad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad \Gamma \vdash \overline{e} : \overline{T'}}{\Gamma \vdash \mathtt{fun}(\overline{e}) : \sigma(T')}$$

$$\text{(T-CASE)}\quad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash \overline{p \Rightarrow e_p} : T \rightarrow T'}{\Gamma \vdash \mathbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} : T'} \qquad \text{(T-BRANCH)}\quad \frac{\Gamma \vdash p : T \qquad \Gamma \circ \text{pmatch}(p, T) \vdash e_p : T'}{\Gamma \vdash p \Rightarrow e_p : T \rightarrow T'} \qquad \text{(T-SUB)}\quad \frac{\Gamma \vdash e : T \qquad T \leq T'}{\Gamma \vdash e : T'}$$

$$\text{(T-FUTGUARD)}\quad \frac{\Gamma \vdash x : \mathtt{Fut}\langle T \rangle}{\Gamma \vdash x? : \mathtt{Bool}} \qquad \text{(T-CRITICGUARD)}\quad \frac{\Gamma \vdash x : (\mathtt{I}, \mathbb{r})}{\Gamma \vdash \|x\| : \mathtt{Bool}} \qquad \text{(T-CONJGUARD)}\quad \frac{\Gamma \vdash g_1 : \mathtt{Bool} \qquad \Gamma \vdash g_2 : \mathtt{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathtt{Bool}}$$

Fig. 2.4 Typing rules for the functional level

of any type T. Rule (T-CONSEXP) states that the application of the constructor `Co` to a list of expressions $\overline{e}$ is of type $\sigma(T')$ whenever the constructor is of a functional type $\overline{T} \rightarrow T'$ and the expressions are of type $\overline{T'}$; where the auxiliary function tmatch applied on the formal types $\overline{T}$ and the actual ones $\overline{T'}$ returns the substitution σ. Rule (T-FUNEXP) is similar to the previous one for constructor expressions, namely, the application of the function `fun` to a list of expressions $\overline{e}$ is of type $\sigma(T')$ whenever the function is of a functional type $\overline{T} \rightarrow T'$ and the expressions are of type $\overline{T'}$, and again tmatch is applied to obtain σ. Rule (T-CASE) states that if all branches in $\overline{p \Rightarrow e_p}$ are well typed with the same type, then the case expression is also well typed with the return type of the branches. Rule (T-BRANCH) states that a branch $p \Rightarrow e_p$ is well typed with an arrow type $T \rightarrow T'$ if the pattern p is well typed with T and the expression e_p is well typed with type T' in the composition of Γ with typing assertions for the pattern obtained by the function pmatch, previously defined. Rule (T-SUB) is the standard subsumption rule, which uses the subtyping relation defined in Sect. 2.2.

Guard Expressions The typing rules for guard expressions are given at the bottom of Fig. 2.4. Rule (T-FUTGUARD) states that if a variable x has type $\mathtt{Fut}\langle T \rangle$, the guard $x?$ has type `Bool`. Rule (T-CRITICGUARD) states that $\|x\|$ has type `Bool` if x is an object, namely having type $(\mathtt{I}, \mathbb{r})$. Rule (T-CONJGUARD) states that if each g_i has type `Bool` for $i = 1, 2$ then the conjunction $g_1 \wedge g_2$ has also type `Bool`.

$$\text{(T-EXP)}\quad \frac{\Gamma \vdash e : T}{\Gamma, \emptyset \vdash e : T} \qquad\qquad \text{(T-GET)}\quad \frac{\Gamma \vdash e : \mathtt{Fut}\langle T\rangle}{\Gamma, \emptyset \vdash \mathbf{get}(e) : T}$$

$$\text{(T-NEW)}\quad \frac{\begin{array}{c}\Gamma(\mathbf{this}) = (\mathtt{C}', \mathtt{G}[\ldots]) \\ \mathit{params}(\mathtt{C}) = \overline{T\ x} \qquad \Gamma \vdash \overline{e : T'} \qquad \mathrm{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad T \in \mathrm{crec}(\mathtt{G}, \mathtt{C}, \sigma)\end{array}}{\Gamma \vdash \mathbf{new}\ \mathtt{C}(\overline{e}) : T}$$

$$\text{(T-NEWCOG)}\quad \frac{\mathit{params}(\mathtt{C}) = \overline{T\ x} \qquad \Gamma \vdash \overline{e : T'} \qquad \mathrm{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad T \in \mathrm{crec}(\mathtt{G}, \mathtt{C}, \sigma)}{\Gamma, \{\sigma(\mathtt{G})\} \vdash \mathbf{new\ cog}\ \mathtt{C}\ (\overline{e}) : T}$$

$$\text{(T-SCALL)}\quad \frac{\begin{array}{c}\mathit{mtype}(\mathtt{m}, \mathtt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T \\ \Gamma \vdash e : (\mathtt{I}, \sigma(\mathbb{r})) \qquad \Gamma \vdash \overline{e} : \overline{\sigma(T)} \qquad \mathrm{coloc}(\sigma(\mathbb{r}), \Gamma(\mathbf{this}))\end{array}}{\Gamma, \sigma(\mathcal{G}) \vdash e.m(\overline{e}) : \sigma(T)}$$

$$\text{(T-ACALL)}\quad \frac{\mathit{mtype}(\mathtt{m}, \mathtt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T \qquad \Gamma \vdash e : (\mathtt{I}, \sigma(\mathbb{r})) \qquad \Gamma \vdash \overline{e} : \overline{\sigma(T)}}{\Gamma, \sigma(\mathcal{G}) \vdash e!m(\overline{e}) : \mathtt{Fut}\langle\sigma(T)\rangle}$$

Fig. 2.5 Typing rules for expressions with side effects

Expressions with Side Effects The typing rules for expressions with side effects are given in Fig. 2.5. As already stated at the beginning of the section, these typing rules are different wrt the typing rules for pure expressions, as they keep track of the new cogs created. Rule (T-EXP) is a weakening rule which asserts that a pure expression e is well typed in a typing context Γ and an empty set of cogs, if it is well typed in Γ. Rule (T-GET) states that $\mathbf{get}(e)$ is of type T, if expression e is of type $\mathtt{Fut}\langle T\rangle$. Rule (T-NEW) assigns type T to the object **new** $\mathtt{C}(\overline{e})$ if the actual parameters have types compatible with the formal ones, by applying function tmatch; the new object and **this** have the same cog $\mathtt{C}$ and the type T belongs to the crec$(\mathtt{G}, \mathtt{C}, \sigma)$ predicate, which means that T is of the form $(\mathtt{I}, \mathtt{G}[\overline{f : \sigma(\mathtt{I}, \mathbb{r})}])$ and implements$(\mathtt{C}, \mathtt{I})$ and σ is obtained by the function tmatch. Rule (T-NEWCOG) is similar to the previous one, except for the creation of a new cog $\mathtt{G}$ where the new object is placed, and hence the group of object **this** is not checked. Rules (T-SCALL) and (T-ACALL) type synchronous and asynchronous method calls, respectively. Both rules use function mtype to obtain the method signature i.e., $(\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T$. The group record $\mathbb{r}$, the parameters types and the return type of the method are the formal ones. In order to obtain the actual ones, we use the substitution σ that maps formal cog names to actual cog names. The callee e has type $(\mathtt{I}, \sigma(\mathbb{r}))$ and the actual parameters $\overline{e}$ have types $\overline{\sigma(T)}$. Finally, the invocations are typed respectively in the substitution $\sigma(T)$ and $\mathtt{Fut}\langle\sigma(T)\rangle$, with T being the formal return type. Rule (T-SCALL) checks

$$\text{(T-SKIP)}\ \frac{}{\Gamma, \emptyset \vdash \textbf{skip}} \qquad \text{(T-SUSPEND)}\ \frac{}{\Gamma, \emptyset \vdash \textbf{suspend}} \qquad \text{(T-DECL)}\ \frac{\Gamma(x) = T}{\Gamma, \emptyset \vdash T\ x} \qquad \text{(T-COMP)}\ \frac{\Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash s_1; s_2}$$

$$\text{(T-ASSIGN)}\ \frac{\Gamma(x) = T \qquad \Gamma, \mathcal{G} \vdash z : T}{\Gamma, \mathcal{G} \vdash x = z} \qquad \text{(T-ASSIGNFIELDR)}\ \frac{\begin{array}{c} x \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \mathcal{G} \vdash z : T \\ \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[x : T, \ldots]) \end{array}}{\Gamma, \mathcal{G} \vdash x = z}$$

$$\text{(T-ASSIGNFIELDBOT)}\ \frac{\begin{array}{c} x \notin \mathsf{dom}(\Gamma) \qquad T\ x \in \mathit{fields}(\mathtt{C}) \\ \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[\overline{x : T}]) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad x \notin \overline{x} \end{array}}{\Gamma, \mathcal{G} \vdash x = z} \qquad \text{(T-AWAIT)}\ \frac{\Gamma \vdash g : \mathtt{Bool}}{\Gamma, \emptyset \vdash \textbf{await}\ g}$$

$$\text{(T-COND)}\ \frac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2} \qquad \text{(T-WHILE)}\ \frac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma, \emptyset \vdash s}{\Gamma, \emptyset \vdash \textbf{while}\ e\ \{\ s\ \}}$$

$$\text{(T-RETURN)}\ \frac{\Gamma \vdash e : T \qquad \Gamma(\textbf{destiny}) = \mathtt{Fut}\langle T \rangle}{\Gamma, \emptyset \vdash \textbf{return}\ e} \qquad \text{(T-REBIND)}\ \frac{\begin{array}{c} T\ x \in \mathit{ports}(\mathtt{I}) \qquad \Gamma \vdash e : (\mathtt{I}, \mathbb{r}) \\ \Gamma, \mathcal{G} \vdash z : T \qquad \mathrm{coloc}(\mathbb{r}, \Gamma(\textbf{this})) \end{array}}{\Gamma, \mathcal{G} \vdash \textbf{rebind}\ e.x = z}$$

$$\text{(T-REBINDBOT)}\ \frac{\begin{array}{c} \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[\overline{x : T}]) \\ T\ x \in \mathit{ports}(\mathtt{I}) \qquad x \notin \overline{x} \qquad \Gamma \vdash e : (\mathtt{I}, \mathbb{r}) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad \mathrm{coloc}(\mathbb{r}, \Gamma(\textbf{this})) \end{array}}{\Gamma, \mathcal{G} \vdash \textbf{rebind}\ e.x = z}$$

Fig. 2.6 Typing rules for statements

whether the group of **this** and the group of the callee coincide, by using the auxiliary function coloc, whether this check is not performed in rule (T-ACALL).

Statements The typing rules for statements are given in Fig. 2.6. Rules (T-SKIP) and (T-SUSPEND) state that **skip** and **suspend** are always well typed. Rule (T-DECL) states that $T\ x$ is well typed if variable x is of type T in Γ. Rule (T-COMP) states that, if s_1 and s_2 are well typed in the same typing context and, like in linear type systems, they use distinct sets of cogs, then their composition is well typed and uses the disjoint union $\uplus$ of the corresponding cogsets. Rule (T-ASSIGN) states the well typedness of the assignment $x = z$ if both x and z have the same type T and the set of cogs is the one corresponding to z. Rule (T-ASSIGNFIELDR) and rule (T-ASSIGNFIELDBOT) deal with the assignment $x = z$ when field x is not present in $\mathsf{dom}(\Gamma)$ and they follow the same idea as rules (T-FIELDR) and (T-FIELDBOT), respectively. The main difference in the premises of rule (T-ASSIGNFIELDR) and rule (T-ASSIGNFIELDBOT) is the fact that in the former rule x is in the record of **this**, whether in the latter rule x is not in the record of **this** but it is a field of the class of **this**. Rule (T-AWAIT) asserts that **await** g is well typed whenever the guard g has type `Bool`. Rules (T-COND) and

(T-Method)
$$\frac{\Gamma, \overline{x} : \overline{\sigma(T)}, \textbf{destiny} : \mathtt{Fut}\langle\sigma(T)\rangle, \textbf{this} : (\mathtt{C}, \sigma(\mathtt{r})), \sigma(\mathcal{G}) \vdash s}{\Gamma \vdash [\textbf{critical}]\ (\mathcal{G}, \mathtt{r})\ T\ \mathtt{m}(\overline{T\ x})\{\ s\ \}\ \mathit{in}\ \mathtt{C}}$$

(T-Class)
$$\frac{\forall \mathtt{I} \in \overline{\mathtt{I}}.\ \text{implements}(\mathtt{C}, \mathtt{I}) \qquad \Gamma, \overline{x} : \overline{T} \vdash \overline{M}\ \mathit{in}\ \mathtt{C}}{\Gamma \vdash \textbf{class}\ \mathtt{C}\ (\overline{T\ x})\ \textbf{implements}\ \overline{\mathtt{I}}\ \{\ \overline{Fl}\ \overline{M}\ \}}$$

(T-Interface)
$$\frac{\forall \mathtt{I}' \in \overline{\mathtt{I}}.\ \text{extends}(\mathtt{I}, \mathtt{I}')}{\emptyset \vdash \textbf{interface}\ \mathtt{I}\ \textbf{extends}\ \overline{\mathtt{I}}\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}$$

Fig. 2.7 Typing rules for declarations

(T-While) are quite standard, except for the presence of the linear set of cog names: the typing of the conditional statement follows the same principle as the composition of statements in rule (T-Comp); the typing of the loop uses instead an empty set of cogs. Rule (T-Return) asserts that **return** e is well typed if expression e has type T whether the variable **destiny** has type $\mathtt{Fut}\langle T\rangle$. Finally, rule (T-Rebind) types the statement **rebind** $e.x = z$ by checking that: (i) x is a port of the right type, (ii) z has the same type as the port, and (iii) the object stored in e and the current one **this** are in the same cog, by using the predicate coloc($\mathtt{r}$, Γ(**this**)). Rule (T-RebindBot) is similar but it deals with the case when x is not present in the record of **this**, namely it is assigned to $\bot$.

Declarations The typing rules for declarations of methods, classes and interfaces are presented in Fig. 2.7. Rule (T-Method) states that method $\mathtt{m}$ is well typed in class $\mathtt{C}$ if the method's body s is well typed in a typing context augmented with the method's typed parameters; **destiny** being of type $\mathtt{Fut}\langle\sigma(T)\rangle$ and **this** being of type $(\mathtt{C}, \sigma(\mathtt{r}))$. A substitution σ is used to obtain the actual values starting from the formal ones. Rule (T-Class) states that a class $\mathtt{C}$ is well typed when it implements all the interfaces $\overline{\mathtt{I}}$ and all its methods are well typed. Finally, rule (T-Interface) states that an interface $\mathtt{I}$ is well typed if it extends all interfaces in $\overline{\mathtt{I}}$.

Remark The typing rule for assignment requires the group of the variable and the group of the expression being assigned to be the same. This restriction applies to rule for rebinding, as well. To see why this is needed let us consider a sequence of two asynchronous method invocations $x!\mathtt{m}()$; $x!\mathtt{n}()$, both called on the same object and both modifying the same field. Say $\mathtt{m}$ does **this**.$\mathtt{f} = \mathtt{z}_1$ and $\mathtt{n}$ does **this**.$\mathtt{f} = \mathtt{z}_2$. Because of asynchronicity, there is no way to know the order in which the updates will take place at runtime. A similar example may be produced for the case of rebinding. Working statically, we can either force the two expressions $\mathtt{z}_1$ and $\mathtt{z}_2$ to have the same group as $\mathtt{f}$, or keep track of all the different possibilities, thus the type system must assume for an expression a set of possible objects it can reduce to. In this work we adopt the former solution, we let the exploration of the latter as a future work.

$$\frac{\begin{array}{c}\text{(T-Rebind)}\\ \Gamma(\textbf{this}) = (\mathtt{Controller}, \mathtt{G}[\ldots]) \qquad (\mathtt{Server}, \mathtt{r})\ s \in \mathit{ports}(\mathtt{Client}) \\ \forall i = 2, ..., n \quad \Gamma \vdash \mathtt{c}_i : (\mathtt{Client}, \mathtt{G}[\ldots, s : (\mathtt{Server}, \mathtt{r})]) \\ \Gamma, \emptyset \vdash s2 : (\mathtt{Server}, \mathtt{r}) \qquad \mathrm{coloc}(\mathtt{G}[\ldots, s : (\mathtt{Server}, \mathtt{r})], \Gamma(\textbf{this}))\end{array}}{\forall i\ \Gamma, \emptyset \vdash \textbf{rebind}\ \mathtt{c}_i.s = s2}$$

Fig. 2.8 Typing the workflow example

We plan to relax this restriction following a similar idea to the one proposed in [51], where a set of groups can be associated to a variable instead of just only one group.

Example Revisited We now recall the example of the workflow given in Figs. 1.10 and 1.11. We show how the type system works on this example: by applying the typing rule for **rebind** we have the derivation in Fig. 2.8 for any clients from $\mathtt{c}_2$ to $\mathtt{c}_n$. Let us now try to typecheck client $\mathtt{c}_1$. If we try to typecheck the rebinding operation, we would have the following typing judgement in the premise of (T-REBIND):

$$\Gamma(\textbf{this}) = (\mathtt{Controller}, \mathtt{G}[...]) \qquad \Gamma, \emptyset \vdash \mathtt{c}_1 : (\mathtt{Client}, \mathtt{G}'[\ldots, s : (\mathtt{Server}, \quad)])$$

But then, the predicate coloc$(\mathtt{G}'[\ldots, s : (\mathtt{Server}, \mathtt{r})], \Gamma(\textbf{this}))$ is false, since equals $(\mathtt{G}, \mathtt{G}')$ is false. Then, one cannot apply the typing rule (T-REBIND), by thus not typechecking **rebind** $\mathtt{c}_1.\mathtt{s} = \mathtt{s2}$, exactly as we wanted.

2.5 Typing Rules for Runtime Configurations

In this section we present the typing rules for runtime configurations, introduced in Sect. 1.2. In order to prove the subject reduction property, typing rules for runtime configurations are needed and are presented in Fig. 2.9.

Runtime typing judgements are of the form $\Delta, \mathcal{G} \vdash_R N$ meaning that the configuration N is well typed in the typing context Δ by using a set $\mathcal{G}$ of new cogs. The (runtime) typing context Δ is an extension of the (compile time) typing context Γ with runtime information about objects, futures and cogs and is formally defined as follows:

$$\Delta ::= \emptyset \mid \Gamma, \Delta \mid \mathtt{o} : (\mathtt{C}, \mathtt{r}), \Delta \mid \mathtt{f} : \mathtt{Fut}\langle T \rangle, \Delta \mid \mathtt{c} : \mathtt{G}, \Delta$$

An object identifier $\mathtt{o}$ is given type $(\mathtt{C}, \mathtt{r})$ where $\mathtt{C}$ is the class the object is instantiating and $\mathtt{r}$ is the group record containing group information about the object itself and the object's fields. A future value $\mathtt{f}$ is assigned type future $\mathtt{Fut}\langle T \rangle$ and a cog identifier $\mathtt{c}$ is assigned a cog name $\mathtt{G}$.

Rules (T-WEAK1), (T-WEAK2) and (T-WEAK3) state respectively that when an expression is of type T in some typing context Γ, then it has the same type in Δ, which is an extension of Γ; and whenever a statement s or a declaration Dl is well

$$
\textsc{(T-Weak1)}\ \frac{\Gamma, \mathcal{G} \vdash z : T \quad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R z : T}
\qquad
\textsc{(T-Weak2)}\ \frac{\Gamma, \mathcal{G} \vdash s \quad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R s}
\qquad
\textsc{(T-Weak3)}\ \frac{\Gamma, \mathcal{G} \vdash Dl \quad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R Dl}
$$

$$
\textsc{(T-State)}\ \frac{\Delta(x) = T \quad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R T\ x\ v}
\qquad
\textsc{(T-Cont)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T\rangle}{\Delta, \emptyset \vdash_R \mathbf{cont}(\mathtt{f})}
$$

$$
\textsc{(T-Future1)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T\rangle \quad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, v)}
\qquad
\textsc{(T-Future2)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T\rangle}{\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, \bot)}
\qquad
\textsc{(T-Process-Queue)}\ \frac{\Delta, \mathcal{G} \vdash_R Q \quad \Delta, \mathcal{G}' \vdash_R Q'}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R Q \cup Q'}
$$

$$
\textsc{(T-Process)}\ \frac{\Delta, \emptyset \vdash_R \overline{T}\ \overline{x}\ \overline{v} \quad \Delta, \overline{x} : \overline{T}, \mathcal{G} \vdash_R s}{\Delta, \mathcal{G} \vdash_R \{\, \overline{T}\ \overline{x}\ \overline{v} \mid s \,\}}
\qquad
\textsc{(T-Config)}\ \frac{\Delta, \mathcal{G} \vdash_R N \quad \Delta, \mathcal{G}' \vdash_R N'}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R N\ N'}
\qquad
\textsc{(T-Cog)}\ \frac{\Delta(\mathtt{c}) = \mathtt{G}}{\Delta, \{\mathtt{G}\} \vdash_R \mathit{cog}(\mathtt{c}, \mathtt{o}_\varepsilon)}
$$

$$
\textsc{(T-Empty)}\ \frac{}{\Delta, \emptyset \vdash_R \epsilon}
\qquad
\textsc{(T-Idle)}\ \frac{}{\Delta, \emptyset \vdash_R \mathbf{idle}}
$$

$$
\textsc{(T-Object)}\ \frac{\begin{array}{cc}\Delta(\mathtt{o}) = (\mathtt{C}, \mathtt{G}[\overline{f : T}]) & \Delta(\mathtt{c}) = \mathtt{G} \\ \mathit{fields}(\mathtt{C}) = \overline{T}\ \overline{f} & \Delta, \overline{f} : \overline{T}, \emptyset \vdash_R \overline{T}\ \overline{f}\ \overline{v} \\ \Delta, \overline{f} : \overline{T}, \mathcal{G} \vdash_R K_{\mathbf{idle}} & \Delta, \overline{f} : \overline{T}, \mathcal{G}' \vdash_R Q\end{array}}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R \mathit{ob}(\mathtt{o}, \overline{T}\ \overline{f}\ \overline{v},\ \mathit{cog}\ \mathtt{c}; \theta, K_{\mathbf{idle}}, Q)}
$$

$$
\textsc{(T-Invoc)}\ \frac{\begin{array}{cc}\mathit{mtype}(\mathtt{m}, \mathtt{C}) = (\mathcal{G}, \mathtt{r})(\overline{T\ x}) \rightarrow T & \Delta(\mathtt{o}) = (\mathtt{C}, \sigma(\mathtt{r})) \\ \Delta(\mathtt{f}) = \mathtt{Fut}\langle \sigma(T)\rangle & \Delta \vdash_R \overline{v} : \overline{\sigma(T)}\end{array}}{\Delta, \sigma(\mathcal{G}) \vdash_R \mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})}
$$

Fig. 2.9 Typing rules for runtime configurations

typed in Γ, then it is also well-typed in Δ, which is an extension of Γ. Rule (T-STATE) asserts that the substitution of variable x with value v is well typed when x and v have the same type T. Rule (T-CONT) asserts that the statement **cont**(f), which is a new statement added to the runtime syntax, is well typed whenever f is a future. Rule (T-FUTURE1) states that the configuration *fut*(f, v) is well typed if the future f has type Fut$\langle T\rangle$ where T is the type of v. Instead, rule (T-FUTURE2) states that *fut*(f, $\bot$) is well typed whenever f is a future. Rule (T-PROCESS- QUEUE) states that the union of two queues is well typed if both queues are well typed and the set of cogs is obtained as a disjoint union of the two sets of cogs corresponding to each queue. Rule (T-PROCESS) states that a task or a process is well typed if its local variables $\overline{x}$ are well typed and statement s is well typed in a typing context augmented with typing information about the local variables and the set of cogs $\mathcal{G}$. Rule (T-CONFIG) states that the composition $N\ N'$ of two configurations is well typed whenever N and N' are well typed using disjoint sets of cog names. Rule (T-COG) asserts that a group configuration cog(c, $\mathtt{o}_\epsilon$) is well typed if c is declared to be associated to G in Δ. Rules (T-EMPTY) and (T-IDLE) are straightforward. Rule (T-OBJECT) states that an object is well typed whenever: (i) the declared record of o is the same as the one associated to c; (ii) its fields are well typed and (iii) its running process and process queue are well typed. Finally, (T-INVOC) states that $\mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$ is well typed under substitution σ when: (i) callee $\mathtt{o}$ is assigned type $(\mathtt{C}, \sigma(\mathtt{r}))$; (ii) future $\mathtt{f}$ is of type $\mathtt{Fut}\langle\sigma(T)\rangle$ and (iii) values $\overline{v}$ are typed accordingly by applying substitution σ, namely $\overline{\sigma(T)}$.