Testing object Interactions

Grüner, A.

Chapter 6

Concurrent programming language – *CoJapl*

6.1 Syntax

As mentioned in the introduction, we incorporate concurrency into the programming language *Japl* of Chapter 2 by means of *thread classes*. The corresponding syntactical modifications are rather straightforward. The grammar for the resulting concurrent *Java*-like language, *CoJapl*, is given in Table 6.1. Once again, to emphasize the extending character, we grayed out the constructs that are inherited from the sequential programming language *Japl*. The grammar shows that a thread class definition resembles the definition of an object class constructor. That is, we do not embark on the strategy of *Java* or C♯, where thread classes are realized by means of designated object classes. Instead, introducing a new kind of classes allows for a clear separation of concerns, as object classes are the generators of *state* while thread classes are used to generate *activity*.

The signature of a thread class provides a thread class name C and a list of formal parameters. The body of a thread class consists of a body statement *stmt* and a concluding `return`. Thus, a *CoJapl* program p does not only provide a sequence of class definitions $\overline{cldef}$ but additional it allows to define a sequence of thread class definitions $\overline{tdef}$.

The counterpart of a thread class definition is the `spawn` statement, which is used to create a new thread instance from a thread class and which, thus, has some similarities with the `new` statement. It specifies the name of the thread class which serves as the code template for the new thread. A sequence of expressions $\overline{e}$ represent the actual parameter of the new thread. The `spawn` statement is an assignment. It allows to store the *thread identifier* of the new thread in a variable x. A thread identifier is comparable with an object name insofar as it uniquely identifies a thread. Note however, that a thread is not allocated on the heap. Specifically, we assume the existence of another infinite set `thread` which serves

p	$::= \overline{impdecl};\ \overline{T}\,\overline{x};\ \overline{cldef}\ \overline{tdef}\ \{\ stmt;\ \texttt{return}\ \}$	program
$impdecl$	$::= \texttt{import}\ C$	import declaration
$cldef$	$::= \texttt{class}\ C\{\ \overline{T}\,\overline{f};\ con\ \overline{mdef}\ \}$	class definition
con	$::= C(\overline{T}\,\overline{x})\{\ \overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ \}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\ \overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\ \}$	meth. definition
$tdef$	$::= \texttt{thread}\ C(\overline{T}\,\overline{x})\{\ stmt;\ \texttt{return}\ \}$	thread class definition
$stmt$	$::= x = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	statements
	$\mid f = e \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
	$\mid x = \texttt{spawn}\ C(e,\ldots,e)$	
e	$::= x \mid f \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(e,\ldots,e) \mid \texttt{tid} \mid \texttt{tclass}$	expressions

Table 6.1: *CoJapl* language: syntax

as the domain of thread identifiers. For the sake of simplicity we do not allow to pass around thread identifiers in terms of a parameter or a return value. Within the thread itself, its thread identifier can be found out by means of the new expression `tid`. Moreover, a thread may also identify the name of its thread class using the expression `tclass`.

6.2 Static semantics

Similar to the syntax definition, also the type system needs only small changes regarding the concurrency extension. Concerning the typing rules for the syntactical constituents up to statements, given in Table 6.2, we only have to modify Rule T-Prog and to add two rules for the two new constructs, namely for thread definitions and for the `spawn` statement. Apart from these changes, we keep the rules from Table 2.2 and, respectively, Table 2.9 without any changes.

As for the Rule T-Prog, we only have change the definition of the commitment context Θ, as not only the object classes but also the thread classes are provided to the program's environment. This is essential as it enables an external component to instantiate a thread class defined in the program. On account of this, we have to extend the definition of the auxiliary function *cltype*. While in the sequential setting the function *cltype* was used in order to extract the typing information of a class from the corresponding object class definition, in the concurrent setting it additionally has to extract the typing information from thread class definitions. Thus, we extend the definition given in Section 2.2 by the following definition:

$$cltype(\ \texttt{thread}\ C(\overline{T}\,\overline{x})\{\ stmt;\ \texttt{return}\}\ ) \stackrel{\text{def}}{=} C : \overline{T}$$

Therefore, in contrast to the type of an object class, a thread class type consists of its parameter type list $\overline{T}$, only. Note, specifically, that a thread class type is not

a functional type because a thread does not provide a return value. Furthermore, note in this context that we use the same domain *CNames* for, both, object classes and thread classes. Hence, we assume all names of object classes *and* thread classes to be unique within the program. In particular, a class name C is at most either typed as an object class or as a thread class.

Rule T-TDef deals with the syntax check of thread class definitions. Again, the rule is almost identical to the corresponding rule for constructors T-Con. The local type context is extended by the formal parameters which is then used to type check the body statement *stmt* of the thread class.

The `spawn` statement is type-checked by means of Rule T-Spawn. Such a statement is well-typed if the variable x is a thread variable and if the class name C, indeed, refers to a thread class definition such that the thread class's formal parameters and the actual parameters match regarding their types.

Table 6.3 deals with the typing rules for expressions. According to Table 2.3, we only add two new typing rules and keep the rest unchanged. Both the new expression, `tid` and `tclass`, are well-typed in any type context, as a statement is always executed in context of a specific thread. Specifically, we will see in the next section concerning the operational semantics that also the main body statement of the program will be provided with a thread identifier n_{main} and a designated thread class name *Main*.

6.3 Operational semantics

As mentioned earlier, thread classes serve as generators of activity. Indeed, the required modifications of the operational semantics due to the introduction of thread classes mostly affect the call stack, as it represents the active code. Recall, that in *Japl* the call stack captures the sequential flow of control by means of a list of activation records. That is, in the sequential setting, a call stack is of the form

$$\mathsf{CS} = \mathsf{AR}_0 \circ \mathsf{AR}_1^b \circ \mathsf{AR}_2^b \ldots \circ \mathsf{AR}_n^b,$$

where the activation records AR_1^b to AR_n^b are either externally or internally blocked. Hence, they are of the form

$$\mathsf{AR}^b ::= (\mu, \texttt{rcv}\ x;\ mc) \mid (\mu, \texttt{rcv}\ x{:}T;\ mc).$$

The topmost activation record AR_0, however, is either currently in execution or it is externally blocked, i.e., in the latter case the program waits for an incoming communication from the environment. Summarizing, we can say that the form of the call stack as well as the rules of the operational semantics of *Japl* allow to reduce only the topmost statement of the topmost activation record, if at all. This way, the sequential flow of control is ensured. In particular, the operational semantics adheres to the order of the sequentially composed statements.

Regarding the multi-threaded setting, it is natural to use the above mentioned call stack mechanism for each thread, as each thread on its own shall adhere

$$[\text{T-Prog}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Theta = \mathit{cltype}(\overline{\mathit{cldef}}), \mathit{cltype}(\overline{\mathit{tdef}}) \quad \Gamma;\Delta \vdash \overline{\mathit{impdecl}} : \mathsf{ok} \quad \Gamma';\Delta,\Theta \vdash \overline{\mathit{cldef}} : \mathsf{ok} \quad \Gamma';\Delta,\Theta \vdash \overline{\mathit{tdef}} : \mathsf{ok} \quad \Gamma';\Delta,\Theta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\,\overline{x};\ \overline{\mathit{cldef}}\ \overline{\mathit{tdef}}\ \{\mathit{stmt};\ \texttt{return}\} : \Theta}$$

$$[\text{T-Import}]\frac{C \in \mathit{dom}(\Delta)}{\Gamma;\Delta \vdash \texttt{import}\ C : \mathsf{ok}}$$

$$[\text{T-Class}]\frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \texttt{this}{:}C \quad \Gamma';\Delta \vdash \mathit{con} : \mathsf{ok} \quad \Gamma';\Delta \vdash \overline{\mathit{mdef}} : \mathsf{ok}}{\Gamma;\Delta \vdash \texttt{class}\ C\{\overline{T}\,\overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \quad \Gamma';\Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash C\ (\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \quad \Gamma';\Delta \vdash \mathit{stmt} : \mathsf{ok} \quad \Gamma';\Delta \vdash e : T}{\Gamma;\Delta \vdash T\ m(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} : \mathsf{ok}}$$

$$[\text{T-TClass}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Gamma';\Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash \texttt{thread}\ C(\overline{T}\,\overline{x})\{\mathit{stmt};\ \texttt{return}\} : \mathsf{ok}}$$

$$[\text{T-VUpd}]\frac{\Gamma;\Delta \vdash e : \Gamma(x)}{\Gamma;\Delta \vdash x = e : \mathsf{ok}} \qquad [\text{T-FUpd}]\frac{\Gamma;\Delta \vdash e : \Gamma(f)}{\Gamma;\Delta \vdash f = e : \mathsf{ok}}$$

$$[\text{T-Call}]\frac{\Gamma;\Delta \vdash e : C \quad \Gamma(x) = \Delta(C)(m).\mathit{ran} \quad \Gamma;\Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom}}{\Gamma;\Delta \vdash x = e.m(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-New}]\frac{\Gamma(x) = C \quad \Gamma;\Delta \vdash \overline{e} : \Delta(C)(C).\mathit{dom}}{\Gamma;\Delta \vdash x = \texttt{new}\ C(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-Seq}]\frac{\Gamma;\Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma;\Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma;\Delta \vdash \mathit{stmt}_1;\ \mathit{stmt}_2 : \mathsf{ok}} \qquad [\text{T-Block}]\frac{\Gamma, \overline{x}{:}\overline{T};\Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash \{\overline{T}\,\overline{x};\ \mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-While}]\frac{\Gamma;\Delta \vdash e : \mathsf{bool} \quad \Gamma;\Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma;\Delta \vdash \texttt{while}\ (e)\ \{\mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-Cond}]\frac{\Gamma;\Delta \vdash e : \mathsf{bool} \quad \Gamma;\Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma;\Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma;\Delta \vdash \texttt{if}\ (e)\ \{\mathit{stmt}_1\}\ \texttt{else}\ \{\mathit{stmt}_2\} : \mathsf{ok}}$$

$$[\text{T-Spawn}]\frac{\Gamma(x) = \mathsf{thread} \quad \Gamma;\Delta \vdash \overline{e} : \Delta(C)}{\Gamma;\Delta \vdash x = \texttt{spawn}\ C(\overline{e}) : \mathsf{ok}}$$

Table 6.2: *CoJapl* language: type system (stmts)

$$[\text{T-Var}]\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T} \qquad [\text{T-Field}]\frac{\Gamma(f) = T}{\Gamma;\Delta \vdash f : T}$$

$$[\text{T-Null}]\ \Gamma;\Delta \vdash \texttt{null} : C \qquad [\text{T-This}]\frac{\Gamma(\texttt{this}) = C}{\Gamma;\Delta \vdash \texttt{this} : C}$$

$$[\text{T-Op}]\frac{\Gamma;\Delta \vdash \overline{e} : \mathit{dom}(\Delta(\texttt{op})) \quad \mathit{ran}(\Delta(\texttt{op})) = T}{\Gamma;\Delta \vdash \texttt{op}(\overline{e}) : T}$$

$$[\text{T-Tid}]\ \Gamma;\Delta \vdash \texttt{tid} : \texttt{thread} \qquad [\text{T-Tclass}]\ \Gamma;\Delta \vdash \texttt{tclass} : \mathit{CNames}$$

Table 6.3: *CoJapl* language: type system (exprs)

to the order of the statements. Therefore, in the concurrent extension of our programming language, we will make use of a *set of call stacks*. To this end, we will use *thread configuration mappings*. A thread configuration mapping **tc** is a function of the type

$$\mathbf{TC} = \texttt{thread} \rightharpoonup (\mathit{CNames} \times \mathsf{CS})$$

which maps a thread identifier to its call stack, if the program configuration contains a thread with the thread identifier. Otherwise the mapping is undefined for this identifier. In addition to the call stack, however, a call stack mapping provides the thread class name of the thread. We use

$$\mathbf{tc}(n).\mathit{tclass} \quad \text{and} \quad \mathbf{tc}(n).\mathit{cs}$$

in order to refer to the thread class and to the call stack of a thread identifier n, respectively. As for other mappings, we denote the thread configuration mapping that results from modifying **tc** by mapping the thread identifier n to the thread class C and call stack CS with

$$\mathbf{tc}[n \mapsto (C, \mathsf{CS})].$$

Recall, that this means either an extension of the original domain of **tc** by the new element n or an update of **tc** concerning the image of t. In the latter case, we often write

$$\mathbf{tc}[n \mapsto \mathsf{CS}] \quad \text{as a short form for} \quad \mathbf{tc}[n \mapsto (\mathbf{tc}(n).\mathit{tclass}, \mathsf{CS})],$$

as the execution of the thread may change its call stack but not its thread class, anyway.

Therefore, a configuration of the multi-threaded language *CoJapl* only differs from configuration of the sequential language *Japl* in that the call stack is replaced

by a thread configuration mapping. We redefine the set of configurations *Conf* to

$$Conf \stackrel{\text{def}}{=} (\mathsf{H} \times \mathsf{V} \times \mathbf{TC}).$$

In our concurrency model, not all threads are executed in parallel, but the operational semantics implements a scheduler allowing only one thread at a time to exercise an undetermined number of computation steps. That is, the execution of threads is interleaved. Note, that we embark on a *preemptive* concurrency model. We do not provide specific language constructs like *wait* or *notify* by which a thread could influence the actual scheduling. In particular, neither can a thread explicitly give away the control to another thread nor can it claim execution time.

Now let us discuss the rules of the operational semantics that deal with internal computation steps. They are defined in Table 6.4 and Table 6.5. Regarding the internal rules, the transition from *Japl*'s sequential setting to *CoJapl*'s multi-threaded setting basically consists of the above mentioned replacement of the call stack by the thread configuration mapping within the configurations. Hence, within each transition rule, the call stack is replaced by a thread configuration mapping. Each rule *non-deterministically* chooses a thread identifier n of the thread configuration mapping's domain. Then the associated call stack is reduced much like in the corresponding rules of the sequential setting. Finally, the resulting thread configuration replaces the original configuration within the thread configuration mapping. Note, non-deterministically choosing a thread represents a very simple scheduling policy which, specifically, does not guarantee *fairness*. In other words, theoretically it may happen that a specific thread never gets any execution time.

As for Rule Call, the transition from *Japl* to *CoJapl* does not only entail the above mentioned call stack replacement, but additionally we have to provide the method with values for the expressions `tid` and `tclass`. In order to find out the thread class of the thread n, we do not only look up the thread's call stack in the thread configuration mapping **tc** but also its thread class C_T.

Regarding Rule Spawn some more words are in order. We assume that a thread with thread identifier n_1 is about to spawn a new thread of thread class C. To this end, we choose a new thread name n_2 which is not already in use, hence, which is not in the domain of the thread configuration mapping **tc**, already. The new thread identifier n_2 is returned to the call stack of thread n_1 which correspondingly updates the value of variable x by modifying the local variable function list and the global variables. As for the new thread, we create a new call stack CS_2 which consists of a single activation record, only. In particular, the activation record code is represented by the body of thread class C and its local variable function list consists of the variable function v_l only, capturing the thread's identifier, its class name, and the parameters $\overline{x}$ of the `spawn` statement. Finally the thread configuration mapping is updated in that, on the one hand, it gets extended regarding thread identifier n_2 and, on the other hand, the entry of thread identifier n_1 is updated.

$$[\textsc{Ass}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, x = e; mc) \circ \mathsf{CS}^b \\ (\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu}) \quad \mathsf{CS} = (\mu', mc) \circ \mathsf{CS}^b\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{FUpd}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, f = e; mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b \\ o = [\![\mathtt{this}]\!]_h^{\mathsf{v},\mu} \quad (C, \mathsf{F}) = h(o) \quad h' = h[o \mapsto (C, \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])]\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h', \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c}\mathbf{tc}(n) = (C_T,\ (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \\ \mathsf{CS} = (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \mathtt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \\ o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(o).class \quad \overline{T}\,\overline{x} = mparams(C, m) \quad \overline{T_l}\,\overline{x_l} = mvars(C, m) \\ \mathsf{v}_l = \{\mathtt{this} \mapsto o, \mathtt{tid} \mapsto n, \mathtt{tclass} \mapsto C_T, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{New}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, x = \mathtt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mathsf{v}_l, cbody(C);\ \mathtt{return\ this}) \circ (\mu, \mathtt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \\ o \in N \setminus dom(h) \quad h' = h[o \mapsto Obj_\perp^C] \quad \overline{T}\,\overline{x} = cparams(C) \quad \overline{T_l}\,\overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\mathtt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h', \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{BlkBeg}]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \{\overline{T}\,\overline{x}; stmt\};\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mathsf{v}_l{\cdot}\mu, stmt;\ \mathtt{BE}\ mc) \circ \mathsf{CS}^b \\ \mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{BlkEnd}]\ \frac{\mathbf{tc}(n).cs = (\mathsf{v}_l{\cdot}\mu, \mathtt{BE}\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Whl}_1]\ \frac{\begin{array}{c}\mathbf{tc}(n).cs = (\mu, \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \\ \mathsf{CS} = (\mu, stmt;\ \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \quad [\![e]\!]_h^{\mathsf{v},\mu}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Whl}_2]\ \frac{\mathbf{tc}(n).cs = (\mu, \mathtt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, mc) \circ \mathsf{CS}^b \quad \neg[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

Table 6.4: *CoJapl* language: operational semantics (internal, part 1)

Also the interface communication rules of the operational semantics basically result from the rules of Table 2.12 by exchanging the configuration's call stack with a thread configuration mapping. Additionally, we extend the communication labels a concerning incoming and outgoing calls and returns with the thread n

$$[\textsc{Cond}_1]\ \frac{\mathbf{tc}(n).cs = (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, stmt_1;\ mc) \circ \mathsf{CS}^b \quad [\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Cond}_2]\ \frac{\mathbf{tc}(n).cs = (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu, stmt_2;\ mc) \circ \mathsf{CS}^b \quad \neg[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}, \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Ret}]\ \frac{\mathbf{tc}(n).cs = (\mu_1, \texttt{return}\ e) \circ (\mu_2, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS} = (\mu_2', mc) \circ \mathsf{CS}^b \quad (\mathsf{v}', \mu_2') = vupd(\mathsf{v}, \mu_2, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu_2})}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n \mapsto \mathsf{CS}])}$$

$$[\textsc{Spawn}]\ \frac{\begin{array}{c}\mathbf{tc}(n_1).cs = (\mu, x = \texttt{spawn}\ C(\overline{e});\ mc) \circ \mathsf{CS}^b \quad \mathsf{CS}_1 = (\mu', mc) \circ \mathsf{CS}^b \\ n_2 \in N \setminus dom(\mathbf{tc}) \quad (\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto n_2) \quad \mathsf{CS}_2 = (\mathsf{v}_l, tbody(C)) \\ \overline{T}\,\overline{x} = tparams(C) \quad \mathsf{v}_l = \{\texttt{tid} \mapsto n_2, \texttt{tclass} \mapsto C, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}\}\end{array}}{(h, \mathsf{v}, \mathbf{tc}) \rightsquigarrow (h, \mathsf{v}', \mathbf{tc}[n_1 \mapsto \mathsf{CS}_1][n_2 \mapsto \mathsf{CS}_2])}$$

Table 6.5: *CoJapl* language: operational semantics (internal, part 2)

that carries out the communication step:

$$a ::= \gamma? \mid \gamma!$$
$$\gamma ::= n\langle call\ o.m(\overline{v})\rangle \mid n\langle new\ C(\overline{v})\rangle \mid n\langle return\ (v)\rangle \mid \nu(\Delta, \Theta).\gamma,$$
where $o \in N, v \in Vals$ and where Δ and Θ are type mappings.

To understand the reason for the extension of the labels, recall that a communication label shall consist of exactly the information that is passed to the receiver by the corresponding communication step. Now if, for instance, an incoming method call occurs, then the program does not only recognize the method m, the callee o, and the actual parameters $\overline{v}$ of the call but it can also find out the corresponding thread identifier by means of the expression `tid`. The same applies to constructor calls and to returns.

Note, in particular, that the thread of an incoming call may show up for the first time. Hence, the thread identifier may be included in the type mapping of the ν-binder. Since the program may inquire the thread class via `tclass`, such a new thread is typed with its class name. In contrast to the ν-binder of the sequential setting, the ν-binder of the multi-threaded setting consists of two mappings, representing the assumed and the committed types. For, in *Japl* an interface communication may only update either the commitment context or the assumption context. In *CoJapl* this is not always the case, anymore. The reason will become clear soon.

Apart from the program configuration modifications and from the above mentioned label extension, we have to deal with a new kind of interface communication, namely *cross-border thread spawning*. More specifically, the program may spawn a thread of an externally defined thread class provoking an *outgoing thread spawn label*. Likewise, the environment may spawn a thread concerning a thread class of the program resulting in an *incoming thread spawn label*. Again, the justification for dedicated labels regarding thread spawning is that the spawn is obviously an observable interaction: the new thread itself is aware of the fact that it just has been spawned. In order to find out the constituents of a spawn label, let us assume that the program spawns a new thread of an externally defined thread class. Such a spawn is certainly implemented in terms of a spawn statement

$$x = \texttt{spawn}\ C(\overline{e}),$$

where we consider C to be an externally defined thread class, i.e., a thread class of the program's environment. Similar to the cross-border constructor call, the name of the class and the actual parameters are part of the communication label. In contrast to a constructor call, where the calling thread is blocked until the environment yields the new object name, a thread spawn immediately returns and, thus, immediately yields the new thread identifier. As a consequence, the outgoing spawn label is equipped with the new thread identifier, such that the communication step provides both the communication partners with the new identifier. Symmetrically, an incoming spawn label includes the new thread identifier, as well. Therefore, we extend the above communication label definition as follows:

$$\gamma ::= \langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle.$$

Note that the spawn label γ provides the identifier n of the newly created thread only but not the thread identifier of the thread that has executed the `spawn` statement, as it is unknown to the new thread and, thus, unknown to the receiver of the communication step.

Now let us get back to the ν-binder. In the sequential setting a new name communicated in terms of an *incoming* communication represents always an object of an *environment* class, that is, the ν-binder of *incoming* communication always consists of an *assumption* type mapping Δ', only – objects of program classes are always created by the program itself. We have just seen, however, that regarding the multi-threaded setting an *incoming spawn* provides the identifier of the new thread already, even though the thread class is part of the *program*. Consequently, the thread identifier of the incoming spawn is typed with the program class such that the ν-binder includes a commitment type mapping Θ'. The parameters e of the spawn, yet again, may entail the propagation of new environment objects as well, thus, the spawn label is equipped with, both, a commitment *and* an assumption type context.

After this general introduction, the interface communcation rules of the operational semantics are given in Table 6.6. Similar to the internal computation

$$[\textsc{SpawnO}]\ \frac{\begin{array}{c} a = \nu(\Theta', \Delta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle! \quad C \in \mathit{dom}(\Delta) \\ \mathbf{tc}(n') = (\mu, x{=}\texttt{spawn}\ C(\overline{e}); mc) \circ \mathsf{CS} \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mu', mc) \circ \mathsf{CS}] \end{array}}{\Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta \vdash (h, \mathsf{v}', \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$,
$n \in N \setminus \mathit{dom}(\mathbf{tc})$,
$(\mathsf{v}', \mu') = \mathit{vupd}(\mathsf{v}, \mu, x \mapsto n)$,
$\Delta' = (n{:}C)$, and
$\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{SpawnI}]\ \frac{\begin{array}{c} a = \nu(\Delta', \Theta').\langle \mathit{spawn}\ n\ \mathit{of}\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (C, (\mathsf{v}_l, \mathit{tbody}(C))] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta, \Theta'}$$

where $\overline{T}\,\overline{x} = \mathit{tparams}(C)$ and
$\mathsf{v}_l = \{\texttt{tid} \mapsto n, \texttt{tclass} \mapsto C,$
$\overline{x} \mapsto \overline{v}\}$

$$[\textsc{CallI}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}(n) = (C_T, \mathsf{CS}^{eb}) \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mathsf{v}_l, \mathit{mbody}(C, m)) \circ \mathsf{CS}^{eb}] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $\Theta \vdash o : C$,
$\overline{T}\,\overline{x} = \mathit{mparams}(C, m)$,
$\overline{T'}\,\overline{x'} = \mathit{mvars}(C, m)$, and
$\mathsf{v}_l = \{\texttt{tid} \mapsto n, \texttt{tclass} \mapsto C_T,$
$\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v},$
$\overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

$$[\textsc{NewI}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{new}\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \mathbf{tc}(n) = (C_T, \mathsf{CS}^{eb}) \\ \mathbf{tc}' = \mathbf{tc}[n \mapsto (\mathsf{v}_l, \mathit{cbody}(C)) \circ \mathsf{CS}^{eb}] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h', \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $o \in N \setminus \mathit{dom}(h)$,
$h' = h[o \mapsto \mathit{Obj}_{\perp}^{C}]$,
$\overline{T}\,\overline{x} = \mathit{cparams}(C)$,
$\overline{T'}\,\overline{x'} = \mathit{cvars}(C)$, and
$\mathsf{v}_l = \{\texttt{tid} \mapsto n, \texttt{tclass} \mapsto C_T,$
$\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v},$
$\overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

$$[\textsc{CallI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{call}\ o.m(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \Delta' \vdash n : C_T \\ \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\ C_T, \mathsf{v}_l, \mathit{mbody}(C, m))\ )\ ] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h, \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $\Theta \vdash o : C$,
$\overline{T}\,\overline{x} = \mathit{mparams}(C, m)$,
$\overline{T'}\,\overline{x'} = \mathit{mvars}(C, m)$, and
$\mathsf{v}_l = \{\texttt{tid} \mapsto n, \texttt{tclass} \mapsto C_T,$
$\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v},$
$\overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

$$[\textsc{NewI}_{nt}]\ \frac{\begin{array}{c} a = \nu(\Delta').n\langle \mathit{new}\ C(\overline{v})\rangle? \quad \Delta \vdash a : \Theta \\ \Delta' \vdash n : C_T \\ \mathbf{tc}' = \mathbf{tc}[\ n \mapsto (\ C_T, (\mathsf{v}_l, \mathit{cbody}(C))\ )\ ] \end{array}}{\Delta \vdash (h, \mathsf{v}, \mathbf{tc}) : \Theta \xrightarrow{a} \Delta, \Delta' \vdash (h', \mathsf{v}, \mathbf{tc}') : \Theta}$$

where $o \in N \setminus \mathit{dom}(h)$,
$h' = h[o \mapsto \mathit{Obj}_{\perp}^{C}]$,
$\overline{T}\,\overline{x} = \mathit{cparams}(C)$,
$\overline{T'}\,\overline{x'} = \mathit{cvars}(C)$, and
$\mathsf{v}_l = \{\texttt{tid} \mapsto n, \texttt{tclass} \mapsto C_T,$
$\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v},$
$\overline{x'} \mapsto \mathit{ival}(\overline{T'})\}$

Table 6.6: *CoJapl* language: operational semantics (external)

rules, most of the external rules are similar to their sequential counterparts of Table 2.12. As mentioned above, however, we additionally introduce two rules concerning incoming spawns and, respectively, outgoing spawns. Rule SpawnO deals with a `spawn` statement that results in an outgoing spawn label, i.e., the corresponding thread class C is in the domain of the assumption context Δ. We said already that the ν-binder of a spawn label provides an assumption and an commitment update context. The commitment context Θ' is determined by means of the auxiliary function new introduced in Section 2.4.3. The assumption context consists exactly of the thread identifier n which is used for the newly spawned thread.

Dually, Rule SpawnI deals with an incoming spawn label. The domain of the thread configuration mapping is extended by the thread class name C and a new call stack consisting of the thread class's thread body.

The environment can create threads by means of externally defined thread classes, hence, the corresponding thread creation process is not observable by the program. As a consequence, an incoming call via a new thread may occur, i.e., a thread which is unknown to the program so far. On account of this, the operational semantics of *CoJapl* provides two rules for incoming method calls and, respectively, for incoming constructor calls. Rules CallI and NewI deal with incoming calls by means of a thread n which is known to the program already. In particular, the rules' original thread configuration mapping **tc** maps n to a thread class C_T and a call stack. Thus, the incoming call causes an extension of the existing call stack by a new activation record.

On the other hand, Rule CallI_{nt} and Rule NewI_{nt} are used for incoming calls via an unknown thread n. The novel character of n is indicated by the fact that n is bound in the communication label a such that is included in the label's type context Δ'. Consequently, the thread configuration mapping is extended by the new thread n, where n is mapped to its thread class and a new call stack consisting of the method or, respectively, the constructor body of the call.

Note, in contrast to the previous incoming communication rules, the three new rules SpawnI, CallI_{nt}, and NewI_{nt} dealing with new threads do not require an externally blocked call stack in the original configuration.

We conclude this section with a definition of the program execution, the initial configurations, and the trace semantics of a *CoJapl* program. It is no big surprise that these definitions resemble the corresponding definitions of the sequential language.

Definition 6.3.1 (Execution, initial configurations, and trace semantics)**:** Let

$$p \equiv \overline{impdecl};\ \overline{T}\,\overline{x};\ \overline{cldef}\ \overline{tdef}\ \{stmt;\ \texttt{return}\}$$

be a syntactically correct and well-typed *CoJapl* program. A *program execution* of p is a finite sequence of reduction steps, according to the rules of Table 6.4, Table 6.5, and

$$[\textsc{Intern}]\ \frac{c \rightsquigarrow^* c'}{\Delta \vdash c : \Theta \overset{\epsilon}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Single}]\ \frac{\Delta \vdash c : \Theta \overset{a}{\rightarrow} \Delta' \vdash c' : \Theta'}{\Delta \vdash c : \Theta \overset{a}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Seqnc}]\ \frac{\Delta \vdash c : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta' \qquad \Delta' \vdash c' : \Theta' \overset{t}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}{\Delta \vdash c : \Theta \overset{st}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}$$

Table 6.7: *CoJapl* language: traces

Table 6.6, starting from an *initial configuration* of the program

$$c_{init}(p) \overset{\text{def}}{=} (\ h_\perp, \{\overline{x} \mapsto ival(\overline{T})\},\\ \{\ n_{main} \mapsto (\ \{\texttt{tid} \mapsto n_{main},\ \texttt{tclass} \mapsto Main\},\ stmt;\ \texttt{return}\ )\ \}\ ),$$

or, respectively,

$$\overline{c_{init}}(p) \overset{\text{def}}{=} (h_\perp, \{\overline{x} \mapsto ival(\overline{T})\},\ \epsilon).$$

Correspondingly, by means of the rules of Table 6.7, we define three semantic functions

$$[\![\cdot]\!]^a_{trace}, [\![\cdot]\!]^p_{trace}, [\![\cdot]\!] : \Delta \vdash p : \Theta \rightharpoonup \mathscr{P}(a^*),$$

such that for $\Delta \vdash p : \Theta$ it is

$$[\![\Delta \vdash p : \Theta]\!]^a_{trace} \overset{\text{def}}{=} \{s \in a^* | \Delta \vdash c_{init}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\},$$
$$[\![\Delta \vdash p : \Theta]\!]^p_{trace} \overset{\text{def}}{=} \{s \in a^* | \Delta \vdash \overline{c_{init}}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\},\ \text{and}$$
$$[\![\Delta \vdash p : \Theta]\!] \overset{\text{def}}{=} [\![\Delta \vdash p : \Theta]\!]^a_{trace} \cup [\![\Delta \vdash p : \Theta]\!]^p_{trace}.$$