A Type Checking Algorithm for Concurrent Object Protocols

A Type Checking Algorithm for Concurrent Object Protocols

Luca Padovani

Università di Torino, Dipartimento di Informatica, Corso Svizzera 185, 10149 Torino (TO), Italy

Abstract

Concurrent objects can be accessed and possibly modified concurrently by several running processes. It is notoriously difficult to make sure that such objects are consistent with – and are used according to – their intended protocol. In this paper we detail a type checking algorithm for concurrent objects protocols that provides automated support for this verification task. We model concurrent objects in the Objective Join Calculus and specify protocols using terms of a Commutative Kleene Algebra. The presented results are an essential first step towards the application of this static analysis technique to real-world programs.

Keywords: Objective Join Calculus, concurrent objects, actors, object protocols, behavioral type checking, type inference, Commutative Kleene Algebra

1. Introduction

The flourishing research on *behavioral types* [25] aims at developing static analysis techniques for ensuring that programmable resources are used according to a given *protocol* expressed as a type. Most of the current research in this field focuses on *session types* [22, 23], a family of behavioral types specifically suited to the description of communication protocols over private communication channels called *sessions* which connect two or more processes. A cornerstone trait of virtually all session type systems, regardless of the number of interacting processes, is that the resources granting access to the session – called *session endpoints* – are meant to be used by a single process at any given time. This assumption is key for the soundness of the approach and is enforced by suitable forms of linearity embedded in the type system. In contrast to this setting, the present work considers *concurrent objects*, namely objects that are accessed and possibly modified concurrently by several processes. Instances of such objects are common in everyday concurrent programming and include locks, barriers, queues and future variables. In fact, every object may turn into a concurrent object depending on the context in which it is used. *Actors* [20, 1] as implemented for example in Erlang [4] or Scala Akka [19] are another popular instance of concurrent objects. Sessions themselves can be considered concurrent objects: interacting processes concurrently access the session through one of its endpoints, each endpoint providing a distinct interface to the session [8, 29, 23].

In previous work, Crafa and Padovani [12] have proposed a behavioral type system ensuring that concurrent objects are consistent with, and are used according to, their protocol specified as a behavioral type. Specifically, the type system checks that, whenever a message is sent to a (concurrent) object, that message is allowed to be sent to (and possibly be processed by) the object taking its state into account. Typical examples of protocol violations are: releasing a lock

that has not been acquired beforehand, removing an element from an empty queue, resolving a future variable twice. Not surprisingly, the type system of Crafa and Padovani is quite different from those based on session types. While session types are structured using choice and *sequential composition* [22, 25], behavioral types for concurrent objects are structured using choice, *concurrent composition* and *unlimited sharing*. Sequential composition is attained by means of explicit continuation passing, taking advantage of higher-order types. Crafa and Padovani [12] prove their type system sound but leave the definition of a corresponding type checking algorithm for future work. The contribution of this paper is an achievement of this pending goal. There are various factors that make the type system of Crafa and Padovani challenging to turn into a type checking algorithm. First, there is a loose correspondence between type connectives and code constructs, meaning that the typing rules rely on substantial amounts of guessing and non-local information for finding the "right" way of typing code. Second, the notion of subtyping (and consequently that of type equivalence) crucially embeds the laws of Commutative Kleene Algebra [10, 24]. It follows that semantically related types can be syntactically very different. Third, the extensive use of explicit continuations leads to a proliferation of higher-order types, to the point that a certain amount of type inference is highly desirable if not outright necessary. The type checking algorithm we present in this paper addresses these challenges and is shown to be correct and complete with respect to Crafa and Padovani's type system.

Structure of the paper. We begin overviewing the model of concurrent objects adopted by Crafa and Padovani [12] and the corresponding type system (Section 2). Then, we present an alternative but equivalent formulation of the typing rules that is more amenable to be realized as a type checking algorithm (Section 3). This reduces the type checking problem to resolving a system of *type constraints*, for which we give a resolution procedure (Section 4). Throughout the paper we use a simple but comprehensive example to illustrate all the phases of the type checking algorithm. A slightly more complex example is presented in Section 5. We conclude with an overview of related work (Section 6) and a few hints at future developments (Section 7). Relatively long proofs have been postponed in Appendix A so that they do not interrupt the flow of the main text. CobaltBlue [30] is a Haskell implementation of the presented type checking algorithm. Its distribution includes the source code, a tutorial introduction to concurrent object protocols and several additional examples.

2. The Behaviorally Typed Objective Join Calculus

Our model of concurrent objects is the Objective Join Calculus [16], a mild extension of the Join Calculus [15] whose underlying computational model is the Chemical Abstract Machine [5]. As discussed by Crafa and Padovani [12] and briefly recalled in Section 6, the Objective Join Calculus is a natural core model of typestate-oriented programming (TSOP) in a concurrent setting. The syntax of the calculus is shown in Table 1 and makes use of infinite sets of *object names*, ranged over by a, b, c, of *variables*, ranged over by x, y, z, and of *message tags*, ranged over by $\mathtt{m}$. *Names*, ranged over by u, are either object names or variables. Hereafter we will make extensive use of the notation $\overline{e}$ for denoting finite, possibly empty sequences of various entities. For instance, we write $\overline{u}$ for denoting the sequence $u_1, \ldots, u_n$ of names where n may be 0 if the sequence is empty. We write $|\overline{e}|$ for the length of $\overline{e}$.

Processes in the Objective Join Calculus may have one of four possible forms. The term `null` represents the idle process that does nothing. The term $u!\mathtt{m}(\overline{u})$ represents the process that (asynchronously) sends the *message* $\mathtt{m}(\overline{u})$ to the object u. The message consists of a tag $\mathtt{m}$ and a

Process	P, Q	::=	$\texttt{null}$	(idle process)
		|	$u!\texttt{m}(\overline{u})$	(message output)
		|	$P \,\&\, Q$	(process composition)
		|	$\texttt{object}\ a : t\ [C]\ P$	(object definition)
Pattern	J, K	::=	$\texttt{m}(\overline{x})$	(message pattern)
		|	$J \,\&\, K$	(pattern composition)
Class	C, D	::=	$J \triangleright P$	(reaction rule)
		|	$C \mid D$	(class composition)

Table 1: Syntax of the behaviorally typed Objective Join Calculus.

possibly empty sequence of arguments $\overline{u}$. The *arity* of the message is the length of $\overline{u}$. The term $P \,\&\, Q$ represents the parallel composition of P and Q. Finally, the process $\texttt{object}\ a : t\ [C]\ P$ creates an object a with type t and scope P. Syntax and semantics of types will be given shortly. The behavior of the object is described by its *class* C, which is a non-empty, finite collection of *reaction rules* of the form $\texttt{m}_1(\overline{x_1}) \,\&\, \cdots \,\&\, \texttt{m}_k(\overline{x_k}) \triangleright Q$. Whenever the messages $\texttt{m}_1(\overline{c_1}), \ldots, \texttt{m}_k(\overline{c_k})$ are targeted to a, they are atomically consumed and the process $Q\{\overline{c_1}/\overline{x_1}\} \cdots \{\overline{c_k}/\overline{x_k}\}$ is spawned, where $P\{c/x\}$ is the usual notation for the capture-avoiding substitution of the free occurrences of x with c in P and $P\{\overline{c}/\overline{x}\}$ its obvious extension to same-length sequences of names and variables.

We do not provide further details on the semantics of processes, which is irrelevant in this paper, and we omit other syntactic forms – most notably molecules and soups – that are needed only to describe their operational semantics. The interested reader may refer to Fournet et al. [16] and Crafa and Padovani [12]. The notions of free and bound names follow from the syntax as expected, noting that the binders are message patterns $\texttt{m}(\overline{x})$, whose scope is the process on the right-hand side of the reaction in which they occur, and object definitions. The name of an object is visible in its own reactions. We identify processes modulo their bound names.

Example 1. We introduce an example to illustrate the features of the Objective Join Calculus and discuss all aspects of the typing discipline and the type checking algorithm. The example shows the modeling of a lock along with two user processes that compete for acquiring it:

$$
\begin{array}{l}
\texttt{object}\ \mathit{lock} : t_{\mathit{lock}}\ [\,\texttt{FREE} \,\&\, \texttt{Acquire}(\mathit{sender}) \triangleright \mathit{lock}!\texttt{BUSY} \,\&\, \mathit{sender}!\texttt{Reply}(\mathit{lock}) \\
\qquad\qquad\qquad\qquad\quad \mid \texttt{BUSY} \,\&\, \texttt{Release} \qquad\quad \triangleright \mathit{lock}!\texttt{FREE}\,] \\
\texttt{object}\ \mathit{user} : t_{\mathit{user}}\ [\,\texttt{Reply}(\mathit{lock}) \triangleright \mathit{lock}!\texttt{Release}\,] \\
\mathit{lock}!\texttt{FREE} \,\&\, \mathit{lock}!\texttt{Acquire}(\mathit{user}) \,\&\, \mathit{lock}!\texttt{Acquire}(\mathit{user})
\end{array}
$$

The *lock* object understands four kinds of messages tagged `FREE`, `BUSY`, `Acquire`, `Release`. The former two messages encode the *state* of the lock, whereas the latter two messages represent its *operations*. The *lock* object has two reactions. The first one fires when the lock is `FREE` and there is a pending `Acquire` operation targeted to it. The `Acquire` message has a *sender* argument representing the process willing to acquire the lock. When the reaction fire, the `FREE` and `Acquire` messages are consumed, the lock turns to state `BUSY` by targeting a `BUSY` message to itself, and *sender* is notified that it has acquired the lock with a `Reply` message. The `Reply` message carries an explicit continuation, that is a reference to *lock* that *sender* will use to release it. The need for this continuation arises from the fact that the protocol (hence the type) of a lock

to be acquired is different from the protocol (hence the type) of a lock to be released. We will say more on this when we discuss typing (Example 3). The second reaction of *lock* fires when the lock is `BUSY` and there is a pending `Release` operation targeted to it. When this happens, the `BUSY` and `Release` messages are consumed and the lock becomes `FREE` again.

The *user* object understands a `Reply` message carrying a reference to a lock. When a `Reply` message is targeted to *user*, its sole reaction fires consuming the message and releasing the lock referred to in its argument.

The last line of the term initializes the lock in the `FREE` state and models the two users competing to `Acquire` the lock. Since there is just one `FREE` message, only one of the two `Acquire` messages will be consumed by the firing of the first reaction of the lock, while the other one remains pending. The object *user* is then notified that the lock has been acquired, and the reaction in *user* releases it. As the lock becomes `FREE` again, the remaining `Acquire` message triggers the first reaction of *lock* once more and the execution continues as before. ■

We now introduce a type system that enforces concurrent object protocols ensuring that:

1. Each concurrent object conforms to its own protocol.
2. Each concurrent object is used by its clients according to its own protocol.

The syntax of types is given below:

$$\textbf{Type} \qquad t, s \ ::= \ \mathbb{0} \ \mid \ \mathbb{1} \ \mid \ \mathtt{m}(\bar{t}) \ \mid \ t + s \ \mid \ t \cdot s \ \mid \ *t$$

The constant $\mathbb{0}$ represents absurd objects: there is no legal way of using such objects and even not using them is disallowed. These properties of $\mathbb{0}$ make it essentially useless from the programmer's standpoint. Nonetheless, $\mathbb{0}$ plays a key role in the internals of the type checker and cannot be omitted altogether. It also plays the role of the maximum element in the partial ordering among types. The constant $\mathbb{1}$ is the protocol of objects that can only be discarded. Any other usage is prohibited. The *message type* $\mathtt{m}(t_1, \ldots, t_n)$ is the protocol that mandates sending a message `m` with n arguments of type $t_1, \ldots, t_n$ respectively. This protocol implies the obligation of sending the message, hence discarding an object with this type is disallowed. Next we have three behavioral connectives. The sum + represents *choice*: an object of type $t + s$ must be used *either* according to t *or* according to s. For example, an object of type $\mathbb{1} + \mathtt{m}$ can be either discarded or used as target for an `m` message. The product $\cdot$ represents *concurrency*: an object of type $t \cdot s$ must be used *both* according to t *and also* according to s, in possibly concurrent ways. So, for example, an object of type $\mathtt{a} \cdot \mathtt{b}$ must be used as target for both `a` and `b` messages. The exponential $*$ represents *unlimited sharing*: an object of type $*t$ can be used any number of times and possibly concurrently, each time according to t. For example, an object of type $*\mathtt{m}$ can be used as target for an arbitrary number of `m` messages.

We follow common conventions on the precedence of the connectives and assume that $*$ binds stronger than $\cdot$ which in turn binds stronger than +. As an example, $*\mathtt{a} \cdot \mathtt{b} + \mathtt{c}$ means $((*\mathtt{a}) \cdot \mathtt{b}) + \mathtt{c}$.

Example 2. We illustrate the type language on the *lock* object that we have discussed in Example 1. It helps to list the legal usages of *lock* in terms of messages targeted to it:

1. The lock must always be either `FREE` or `BUSY`.
2. There can be any number of users attempting to `Acquire` the lock at any given time.
3. If the lock is `BUSY`, the one client owning it must eventually `Release` it exactly once.

Even if we have not formalized the semantics of types yet, we can intuitively think of a type as of a regular expression generating the allowed message combinations that can be targeted to an object with that type. In the case of *lock*, we can turn the informal protocol description given above into a precise specification by means of the following type:

$$t_{lock} \stackrel{\text{def}}{=} *\texttt{Acquire(Reply(Release))} \cdot (\texttt{FREE} + \texttt{BUSY} \cdot \texttt{Release})$$

The + separating `FREE` and `BUSY` formalizes requirement 1 and imposes that one of these two messages should always be targeted to *lock* so as to indicate in which state *lock* is. The product $\texttt{BUSY} \cdot \texttt{Release}$ formalizes requirement 3 imposing that, when the lock is `BUSY`, we expect one (and only one) `Release` message to be also targeted to *lock*. The type does not specify *when* the `Release` message should be sent, only that it must be eventually sent. The type system will then make sure that, as long as no `Release` message is sent, the process that has acquired the lock has a pending obligation to release it. The exponential in front of `Acquire` and the product that combines it with the rest of the type formalize requirement 2. In particular, the fact that this product combines `Acquire` with the sum of `FREE` and `BUSY` means that `Acquire` messages are allowed to be targeted to *lock* regardless of its state, as we expect. On the contrary, having combined `Release` with `BUSY` means that `Release` message is allowed only when the *lock* is `BUSY`. The message type `Acquire(Reply(Release))` also specifies that the argument carried by the `Acquire` message (*sender* in Example 1) must be used for sending exactly one `Reply` message and that this message, in turn, carries an argument (the rightmost occurrence of *lock* in the first reaction of Example 1) that must be used for sending a `Release` message. ■

We complete the presentation of types with a few conventions. We let `M` range over *message types* of the form $\texttt{m}(\tilde{t})$ and we say that $|\tilde{t}|$ is the arity of $\texttt{m}(\tilde{t})$. Also, we can interpret the productions for types coinductively to allow the generation of possibly infinite types describing possibly infinite protocols, with two assumptions: first, types are *regular*, namely their tree representation must contain finitely many distinct sub-trees; second, types are *contractive* in the sense that every infinite branch in their tree representation must go through infinitely many message types. Regularity ensures that types can be finitely represented [11]. Contractivity prevents the creation of degenerate types like those satisfying the equations $t = t + t$ or $t = *t$. Two types are considered equal if so are their corresponding tree representations.

We now introduce a few auxiliary notions leading to the definition of *subtyping*, which is a cornerstone feature of our type system and therefore of the type checking algorithm. We start from the signature of a type, collecting all of its top-level message types:

Definition 1 (signature)**.** The *signature* of a type t, denoted by $\mathsf{sig}(t)$, is inductively defined by:

$$\mathsf{sig}(\mathbb{0}) = \mathsf{sig}(\mathbb{1}) \stackrel{\text{def}}{=} \emptyset \quad \mathsf{sig}(\texttt{M}) \stackrel{\text{def}}{=} \{\,\texttt{M}\,\} \quad \mathsf{sig}(t + s) = \mathsf{sig}(t \cdot s) \stackrel{\text{def}}{=} \mathsf{sig}(t) \cup \mathsf{sig}(s) \quad \mathsf{sig}(*t) \stackrel{\text{def}}{=} \mathsf{sig}(t)$$

Notice that the signature of a type is well defined also when the type is infinite thanks to the contractivity restriction. Next are the valid configurations of a type t, which are multisets of tags expressing which combinations of messages are legal according to t:

Definition 2 (valid configurations)**.** *Configurations*, ranged over by A, B, are multisets of tags written $\langle \texttt{m}_1, \ldots, \texttt{m}_n \rangle$. The *valid configurations* of a type t, written $[\![t]\!]$, are inductively defined by

$$[\![\mathbb{0}]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![t + s]\!] \stackrel{\text{def}}{=} [\![t]\!] \cup [\![s]\!] \qquad [\![\texttt{m}(\tilde{t})]\!] \stackrel{\text{def}}{=} \{\,\langle \texttt{m} \rangle\,\}$$
$$[\![\mathbb{1}]\!] \stackrel{\text{def}}{=} \{\,\langle\rangle\,\} \qquad [\![t \cdot s]\!] \stackrel{\text{def}}{=} \{\,\mathsf{A} \uplus \mathsf{B} \mid \mathsf{A} \in [\![t]\!], \mathsf{B} \in [\![s]\!]\,\} \qquad [\![*t]\!] \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} [\![t^i]\!]$$

where $\uplus$ denotes multiset sum, $t^0 \stackrel{\text{def}}{=} \mathbb{1}$ and $t^{i+1} \stackrel{\text{def}}{=} t \cdot t^i$ for every $i \in \mathbb{N}$.

Again the set of valid configurations of a type is well defined thanks to the contractivity restriction on types. We can use configurations to distinguish *usable types*, those describing a protocol that can be adhered to, from *unusable types*, those describing the absurd protocol. Examples of unusable types are $\mathbb{0}$ and $\mathbb{0} \cdot t$.

Definition 3 (usable type)**.** We say that t is *usable*, and we write $\mathsf{usable}(t)$, if $[\![t]\!] \neq \emptyset$.

We can now define subtyping:

Definition 4 (subtyping)**.** Let $\leqslant$ be the largest relation on types such that $t \leqslant s$ implies:

1. $[\![s]\!] \subseteq [\![t]\!]$, and
2. for every $\mathtt{m}(s_1, \ldots, s_n) \in \mathsf{sig}(s)$ there exists $\mathtt{m}(t_1, \ldots, t_n) \in \mathsf{sig}(t)$, and
3. for every $\mathtt{m}(s_1, \ldots, s_n) \in \mathsf{sig}(s)$ and $\mathtt{m}(t_1, \ldots, t_n) \in \mathsf{sig}(t)$ and $1 \leq i \leq n$ we have $s_i \leqslant t_i$.

We say that t is a *subtype* of s, and that s is a *supertype* of t, if $t \leqslant s$ holds. We let $\simeq \stackrel{\text{def}}{=} \leqslant \cap \leqslant^{-1}$.

It is a standard property of subtyping relations for object-oriented languages that smaller objects understand a superset of messages than larger objects. This is expressed in clause 2, requiring each message type occurring at the top level in the larger type to also occur at the top level in the smaller type, and with the same arity. Clause 3 expresses the standard contravariant property for argument types, which is best illustrated by the usual safe substitution principle, assuming to have basic types $\mathtt{int} \leqslant \mathtt{real}$. Then, the relation $\mathtt{m(real)} \leqslant \mathtt{m(int)}$ formalizes the intuition that an object of type $\mathtt{m(int)}$, which accepts an $\mathtt{m}$ message with an argument of type $\mathtt{int}$, can be safely replaced by an object of type $\mathtt{m(real)}$, which accepts the same message but can deal with a superset of arguments. Clause 1 expresses the property that, when $t \leqslant s$, the object of type t accepts at least all the message combinations that can be targeted to an object of type s. For example, we have $\mathtt{a}+\mathtt{b} \leqslant \mathtt{a}$. An object of type $\mathtt{a}$, which only accepts a message $\mathtt{a}$, can be safely replaced by an object of type $\mathtt{a}+\mathtt{b}$, which accepts both $\mathtt{a}$ and $\mathtt{b}$ messages. The subtyping relation essentially includes all the laws of Commutative Kleene Algebra such as commutativity of $+$ and $\cdot$, idempotency of $+$, distributivity of $\cdot$ over $+$, identity of $\mathbb{1}$ for $\cdot$ [10]. Other notable relations that we will use in the following are $*t \leqslant \mathbb{1} + t$ and $*t \leqslant *t \cdot *t$. Most importantly, it is easy to see that $\leqslant$ is a pre-congruence with respect to all the type connectives [12].

It should be noted that $\leqslant$ defined here differs from – but is included in – the one given by Crafa and Padovani [12]. The alternative formulation has been chosen because it allows for a technically manageable formalization of the type checking algorithm without changing the properties of $\leqslant$ in a substantial way. One difference concerns unusable types: we have $\mathbb{0} \cdot \mathtt{M} \leqslant \mathbb{0}$ and $\mathbb{0} \not\leqslant \mathbb{0} \cdot \mathtt{M}$, whereas the two types are equivalent according to Crafa and Padovani [12]. This difference has no practical impact, since unusable types do not specify any interesting protocol by definition. The second difference is that $\leqslant$ has limited support for state-dependent message types. For example, $\mathtt{m}_1 \cdot \mathtt{m(a)} + \mathtt{m}_2 \cdot \mathtt{m(b)} \not\leqslant \mathtt{m}_1 \cdot \mathtt{m(a)}$ because $\mathtt{a} \not\leqslant \mathtt{b}$, whereas it is safe to relate the two types. Although we describe the type checking algorithm using Definition 4, CobaltBlue [30] implements a more general one that uses $\leqslant$ as defined by Crafa and Padovani [12].

Let us move on to the typing rules. As usual, we need *type environments* to keep track of the type of free names. A type environment Γ is a partial function from names to types written as either $\overline{u : t}$ or $u_1 : t_1, \ldots, u_n : t_n$. We write $\mathsf{dom}(\Gamma)$ for the domain of Γ, $\Gamma(u)$ for the type associated with u in Γ, $-$ for the empty type environment and Γ_1, Γ_2 for the union of Γ_1 and Γ_2

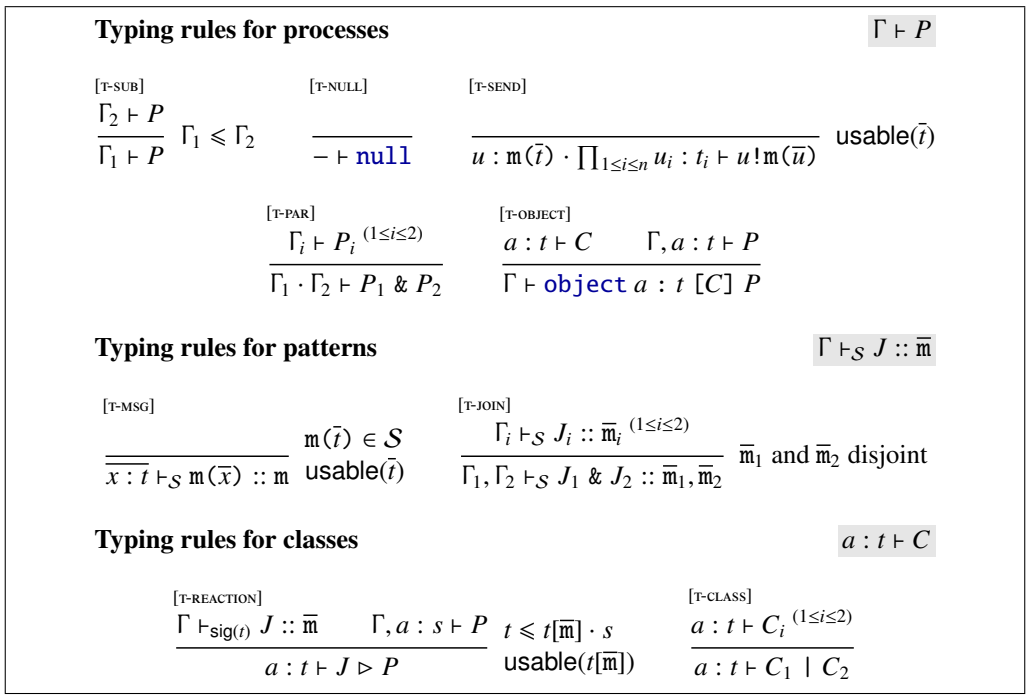

Table 2: Typing rules.

when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We extend the $\cdot$ connective to type environments, thus:

$$(\Gamma_1 \cdot \Gamma_2)(u) \stackrel{\text{def}}{=} \begin{cases} \Gamma_1(u) & \text{if } u \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2) \\ \Gamma_2(u) & \text{if } u \in \mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1) \\ \Gamma_1(u) \cdot \Gamma_2(u) & \text{if } u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The typing rules are shown in Table 2 and derive judgments for processes, patterns and classes. A judgment of the form $\Gamma \vdash P$ states that P is well typed in Γ. Overall, the typing rules for processes are pretty straightforward and check whether P uses the free names occurring in it as specified by their type in Γ. The idle process uses no names and does nothing, therefore it is well typed in the empty type environment. An output process $u!\mathtt{m}(\overline{u})$ uses u for sending an $\mathtt{m}$-tagged message with arguments $u_1, \ldots, u_n$. Note that the type environment is the product of $n + 1$ singleton type environments, each concerning u or one of the u_i. This accounts for the possibility that the same name occurs several times, as in $u!\mathtt{m}(u)$. The usability side condition is necessary for the soundness of the type system [12]. The parallel composition $P_1 \ \& \ P_2$ uses its free names according to the combination of the uses in P_1 and P_2 separately, as expected. The definition of an object a introduces a in the environment when typing the continuation and checks that a is used according to the type annotation t. The class C must conform with t too; we shall discuss the related typing rules shortly. The rule [T-SUB] allows for a form of subsumption: a process that is well typed in an environment Γ_2 is also well typed in an environment Γ_1 whose types are subtypes of those in Γ_2. Formally, the relation $\leqslant$ on type environments is defined thus:

Definition 5 (subtyping on type environments)**.** We write $\Gamma_1 \leqslant \Gamma_2$ if:

1. $\mathsf{dom}(\Gamma_2) \subseteq \mathsf{dom}(\Gamma_1)$, and
2. $\Gamma_1(u) \leqslant \Gamma_2(u)$ for every $u \in \mathsf{dom}(\Gamma_2)$, and
3. $\Gamma_1(u) \leqslant \mathbb{1}$ for every $u \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2)$.

Reading the typing rule [T-SUB] bottom-up, the last clause of Definition 5 allows us to remove names from the type environments when their type t specifies that not using them ($t \leqslant \mathbb{1}$) is legal.

A judgment of the form $\Gamma \vdash_{\mathcal{S}} J :: \mathsf{A}$ states that the pattern J in a reaction for an object whose type has signature $\mathcal{S}$ binds the arguments in Γ and contains message tags in A. The side conditions of rule [T-MSG] makes sure that the message in the pattern is indeed part of the signature and that its arguments have a usable type. The rule [T-JOIN] ensures linearity of tags (each tag occurs at most once in a pattern) and of argument names (each variable is bound at most once in a pattern).

A judgment of the form $a : t \vdash C$ states that the class C for the definition of object a conforms with its type t. The rules [T-REACTION] and [T-CLASS] verify that each reaction maintains the multiset of messages targeted to an object in one of its valid configurations. To grasp the essence of the rules, recall that a reaction $J \triangleright P$ consumes a multiset of messages non-deterministically chosen among those targeted to a that match the pattern J. When the reaction fires, the spawned process P uses the arguments received from the consumed messages and possibly a, where "using a" means, as discussed earlier, producing new messages targeted to a. Rule [T-REACTION] checks that the combination of messages targeted to a *after* the reaction has fired is described by t, the type of a. In order to do so, the rule must take into account which messages remain after those that trigger the reaction have been consumed and which messages are produced. The messages being consumed are those with tags $\overline{\mathtt{m}}$, which correspond to the messages in the pattern J. The messages being produced correspond to a valid configuration of s, which is the type according to which the spawned process P uses a after the reaction has fired. Therefore, the overall combination of messages after the reaction has fired is described by the type $t[\overline{\mathtt{m}}] \cdot s$ where $t[\overline{\mathtt{m}}]$, which will be formalized in a moment, describes the combination of messages remaining by "removing" those with tags $\overline{\mathtt{m}}$ from t. The side condition $t \leqslant t[\overline{\mathtt{m}}] \cdot s$ checks that the new combination of messages targeted to a corresponds to a valid configuration of t, whereas $\mathsf{usable}(t[\overline{\mathtt{m}}])$ makes sure that $\overline{\mathtt{m}}$ is (included in) a valid message configuration for t. The "residual" of a type is formalized thus:

Definition 6 (tag differential)**.** A function $\mathbf{f}$ on types is called a *differential operator* [24] if

$$\mathbf{f}(\mathbb{0}) = \mathbf{f}(\mathbb{1}) = \mathbb{0} \qquad \mathbf{f}(t + s) = \mathbf{f}(t) + \mathbf{f}(s) \qquad \mathbf{f}(t \cdot s) = t \cdot \mathbf{f}(s) + \mathbf{f}(t) \cdot s \qquad \mathbf{f}(*t) = \mathbf{f}(t) \cdot *t$$

for all t and s. The *tag differential* of t with respect to $\mathtt{m}$, denoted by $t[\mathtt{m}]$, is the unique differential operator $\mathbf{f}$ such that $\mathbf{f}(\mathtt{m}(\tilde{t})) = \mathbb{1}$ and $\mathbf{f}(\mathtt{m}'(\tilde{t})) = \mathbb{0}$ if $\mathtt{m} \neq \mathtt{m}'$. We generalize the tag differential to sequences of tags so that $t[\mathtt{m}_1, \ldots, \mathtt{m}_n] \stackrel{\text{def}}{=} t[\mathtt{m}_1] \cdots [\mathtt{m}_n]$.

An alternative intuition for the tag differential is in terms of valid message configurations: if t is the type of an object and there is already an $\mathtt{m}$-tagged message targeted to the object, then the object is meant to be used as described by $t[\mathtt{m}]$. Note that the tag differential is nothing but Brzozowski's derivative [7] adapted to a Commutative Kleene Algebra.

Soundness of the type system (with the generalization of subtyping as discussed after Definition 4) and other standard results such as type preservation have been proved by Crafa and Padovani [12] and will not be repeated here.

Example 3. Let us show that the term described in Example 1 is well typed, where t_{lock} is the type defined in Example 2 and $t_{user} \stackrel{\text{def}}{=} *\texttt{Reply(Release)}$. Starting from the pattern in the first reaction of *lock*, we obtain the derivation $\mathcal{D}_1$

$$\dfrac{\dfrac{}{- \vdash_{\mathcal{S}} \texttt{FREE} :: \texttt{FREE}}\,[\text{T-MSG}] \qquad \dfrac{}{\mathit{sender} : \texttt{Reply(Release)} \vdash_{\mathcal{S}} \texttt{Acquire}(\mathit{sender}) :: \texttt{Reply}}\,[\text{T-MSG}]}{\mathit{sender} : \texttt{Reply(Release)} \vdash_{\mathcal{S}} \texttt{FREE} \,\&\, \texttt{Acquire}(\mathit{sender}) :: \texttt{FREE} \cdot \texttt{Reply}}\,[\text{T-JOIN}]$$

where $\mathcal{S} = \mathsf{sig}(t_{lock}) = \{\texttt{FREE}, \texttt{BUSY}, \texttt{Acquire(Reply(Release))}, \texttt{Release}\}$. We see that *sender* must accept a `Reply` message carrying a reference to another object (the *lock*) with type `Release`. Thus, *sender* is mandated to (eventually) release the lock it has acquired. For the body of the reaction we obtain the derivation $\mathcal{D}_2$

$$\dfrac{\dfrac{}{\mathit{lock} : \texttt{BUSY} \vdash \mathit{lock}!\texttt{BUSY}} \qquad \dfrac{}{\mathit{sender} : \texttt{Reply(Release)}, \mathit{lock} : \texttt{Release} \vdash \mathit{sender}!\texttt{Reply}(\mathit{lock})}}{\mathit{sender} : \texttt{Reply(Release)}, \mathit{lock} : \texttt{BUSY} \cdot \texttt{Release} \vdash \mathit{lock}!\texttt{BUSY} \,\&\, \mathit{sender}!\texttt{Reply}(\mathit{lock})}$$

with two applications of [T-SEND] and one application of [T-PAR]. Using $t_{lock} \leqslant t_{lock}[\texttt{FREE} \cdot \texttt{Acquire}] \cdot \texttt{BUSY} \cdot \texttt{Release}$ we now derive

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\mathit{lock} : t_{lock} \vdash \texttt{FREE} \,\&\, \texttt{Acquire}(\mathit{sender}) \triangleright \mathit{lock}!\texttt{BUSY} \,\&\, \mathit{sender}!\texttt{Reply}(\mathit{lock})}\,[\text{T-REACTION}]$$

that is the first reaction of *lock* is well typed. For the second reaction we obtain instead

$$\dfrac{\dfrac{\vdots}{- \vdash_{\mathcal{S}} \texttt{BUSY} \,\&\, \texttt{Release} :: \texttt{BUSY} \cdot \texttt{Release}}\,[\text{T-JOIN}] \qquad \dfrac{}{\mathit{lock} : \texttt{FREE} \vdash \mathit{lock}!\texttt{FREE}}\,[\text{T-SEND}]}{\mathit{lock} : t_{lock} \vdash \texttt{BUSY} \,\&\, \texttt{Release} \triangleright \mathit{lock}!\texttt{FREE}}\,[\text{T-REACTION}]$$

using $t_{lock} \leqslant t_{lock}[\texttt{BUSY}\cdot\texttt{Release}]\cdot\texttt{FREE}$. Overall we conclude that the definition of *lock* conforms with its type t_{lock}, for every reaction maintains the state of the object into one of the configurations allowed by t_{lock}. Analogous derivations can be used to show that *user* conforms with t_{user}.

Concerning the last line of code we first derive $\mathcal{D}_3$

$$\dfrac{\dfrac{}{\mathit{lock} : t_{\mathtt{A}}, \mathit{user} : \texttt{Reply(Release)} \vdash \mathit{lock}!\texttt{Acquire}(\mathit{user})}\,[\text{T-SEND}]}{\mathit{lock} : *t_{\mathtt{A}}, \mathit{user} : t_{user} \vdash \mathit{lock}!\texttt{Acquire}(\mathit{user})}\,[\text{T-SUB}]$$

where $t_{\mathtt{A}} \stackrel{\text{def}}{=} \texttt{Acquire(Reply(Release))}$ thanks to the property $*t_{\mathtt{A}} \leqslant t_{\mathtt{A}}$, allowing us to eventually use a shareable object of type $*t_{\mathtt{A}}$ once, according to the type $t_{\mathtt{A}}$. We conclude

$$\dfrac{\dfrac{\dfrac{}{\mathit{lock} : \texttt{FREE} \vdash \mathit{lock}!\texttt{FREE}}\,[\text{T-SEND}] \qquad \dfrac{\mathcal{D}_3 \qquad \mathcal{D}_3}{\mathit{lock} : *t_{\mathtt{A}} \cdot *t_{\mathtt{A}}, \mathit{user} : t_{user} \cdot t_{user} \vdash \mathit{lock}!\texttt{Acquire}(\mathit{user}) \,\&\, \cdots}\,[\text{T-PAR}]}{\mathit{lock} : \texttt{FREE} \cdot *t_{\mathtt{A}} \cdot *t_{\mathtt{A}}, \mathit{user} : t_{user} \cdot t_{user} \vdash \mathit{lock}!\texttt{FREE} \,\&\, \mathit{lock}!\texttt{Acquire}(\mathit{user}) \,\&\, \cdots}\,[\text{T-PAR}]}{\mathit{lock} : t_{lock}, \mathit{user} : t_{user} \vdash \mathit{lock}!\texttt{FREE} \,\&\, \mathit{lock}!\texttt{Acquire}(\mathit{user}) \,\&\, \mathit{lock}!\texttt{Acquire}(\mathit{user})}\,[\text{T-SUB}]$$

using the properties $*t_{\mathtt{A}} \leqslant *t_{\mathtt{A}} \cdot *t_{\mathtt{A}}$ and $t_{user} \leqslant t_{user} \cdot t_{user}$, which allow us to share multiple references to *user* and *lock* among different processes without limitations. ■

The previous example illustrates the difficulties encountered in turning the rules in Table 2 into an algorithm. There are two strategies one might attempt. The first one effectively corresponds to a type inference problem, whereby we are given a process P and we try to synthesize a Γ working out a typing derivation for $\Gamma \vdash P$ top down, from the atomic sub-terms of P. Unfortunately, the local structure of P does not always contain enough information to determine precisely how the occurrence of a given identifier should be typed. This is clear looking at the message outputs such as *sender*`!Reply(`*lock*`)` or *lock*`!Acquire(`*user*`)`: the type of the arguments *lock* and *user* depends on how *lock* and *user* will be used by the *receivers* of these messages and there is no hint to the fact that `Release` and `Reply(Release)` are the "right" types for these occurrences of *lock* and *user*. The second strategy corresponds to a classic type checking problem, whereby we are given a process *P and* a type environment Γ and we try to build a typing derivation bottom up, starting from the conclusion $\Gamma \vdash P$. In this case, the rules [T-PAR] and [T-SUB] do not provide any locally available information as to how Γ should be rearranged from the conclusion to the premises of each rule. In the case of [T-SUB], the difficulty is worsened by the fact that the rule is structural and can be applied anywhere in a typing derivation.

3. Generation Rules

In this section we present a set of *generation rules* that can be more easily turned into a type checking algorithm and that we prove to be equivalent to the typing rules of the previous section. The overall strategy for defining these rules is quite standard and based on three main ideas:

- Instead of checking whether a process is well typed with respect to a given type environment, the rules *generate* the type environment in which that process can be well typed.
- Wherever there is not enough information to generate the type environment completely, the rules use *type variables* standing for unknown types to be determined later (Section 4).
- Subtyping relations, which cannot be verified right away because they involve types with type variables, are accumulated in a set of *type constraints* and checked later (Section 4).

From this informal description of the generation rules it is clear that we need to introduce a few new ingredients. We will need an infinite supply of *type variables* ranged over by α, β, γ, which we will use to denote (partially) unknown types. More specifically, we introduce *type expressions*, *guarded expressions* and *argument expressions* as by the following grammar:

$$\begin{array}{rrcl} \textbf{Type Expression} & \tau, \sigma & ::= & \mathbb{0} \mid \mathbb{1} \mid \mathtt{m}(\overline{\omega}) \mid \tau + \sigma \mid \tau \cdot \sigma \mid *\tau \mid \alpha \\ \textbf{Guarded Expression} & f, g & ::= & \mathbb{0} \mid \mathbb{1} \mid \mathtt{m}(\overline{\omega}) \mid f + g \mid f \cdot g \mid *g \\ \textbf{Argument Expression} & \omega & ::= & g \mid \alpha \end{array}$$

As for types, these productions are interpreted coinductively to generate possibly infinite expressions with the same regularity and contractivity restrictions discussed earlier. We write $\mathsf{tv}(\tau)$ for the set of type variables occurring in τ and $\mathsf{utv}(\tau)$ for the set of *unguarded type variables* occurring in τ. In particular, $\mathsf{utv}(\tau)$ is defined by induction on the structure of τ, thus:

$$\mathsf{utv}(\alpha) \stackrel{\text{def}}{=} \{\alpha\} \qquad \mathsf{utv}(\tau + \sigma) = \mathsf{utv}(\tau \cdot \sigma) \stackrel{\text{def}}{=} \mathsf{utv}(\tau) \cup \mathsf{utv}(\sigma)$$
$$\mathsf{utv}(\mathbb{0}) = \mathsf{utv}(\mathbb{1}) = \mathsf{utv}(\mathtt{M}) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{utv}(*\tau) \stackrel{\text{def}}{=} \mathsf{utv}(\tau)$$

Regularity ensures that $\mathsf{tv}(\tau)$ is always finite and contractivity ensures that $\mathsf{utv}(\tau)$ is well defined. Note that types are also guarded expressions and that guarded and argument expressions

are also type expressions. Types, type expressions, guarded expressions and argument expressions differ on whether and where type variables may occur: a type t contains no type variables at all ($\mathsf{tv}(t) = \emptyset$); a type expression may have type variables anywhere; a guarded expression g cannot have unguarded type variables ($\mathsf{utv}(g) = \emptyset$); an argument expression is somewhat in between and can either be a single (unguarded) type variable or a guarded expression. We extend the signature function to type expressions ($\mathsf{sig}(\alpha) \stackrel{\text{def}}{=} \emptyset$) and the usability predicate to guarded expressions ($\mathsf{usable}(\alpha)$ always holds). We also extend the tag differential (Definition 6) to guarded expressions in the obvious way, as guarded expressions have no unguarded type variables. We will see the importance of guarded expressions shortly, when discussing the generation rule [G-REACTION].

Type constraints represent relations that are supposed to hold in order for the process to be well typed. They are generated by the following grammar:

$$\textbf{Constraint} \qquad \varphi \ ::= \ \bot \ \mid \ \omega \leqslant \tau \ \mid \ \mathcal{S} \sqsubset \tau$$

The constraint $\bot$ is unsatisfiable. A constraint of the form $\omega \leqslant \tau$ represents a subtyping relation between two types. We say that constraints of the form $g \leqslant \tau$ are *lower bounds* (g provides information "from below" on the unguarded type variables of τ) and that constraints of the form $\alpha \leqslant \tau$ are *upper bounds* (τ provides information "from above" about the type variable α). A constraint of the form $\mathcal{S} \sqsubset \tau$ relates a signature $\mathcal{S}$ with a type expression τ indicating that each message type in τ is supposed to match at least one in $\mathcal{S}$ with the same tag and arity. We now make these intuitions precise:

Definition 7 (type map)**.** A *type map* $\mathbf{s}$ is a partial function from type variables to types.

As usual, we write $\mathsf{dom}(\mathbf{s})$ for the domain of $\mathbf{s}$, $\mathbf{s}(\alpha)$ for the type associated with α in $\mathbf{s}$ and $\mathbf{s}(\tau)$ for the homomorphic extension of $\mathbf{s}$ to type expressions. If $\mathsf{tv}(\tau) \not\subseteq \mathsf{dom}(\mathbf{s})$, then $\mathbf{s}(\tau)$ is undefined. We extend $\leqslant$ to type maps so that $\mathbf{s}' \leqslant \mathbf{s}$ if and only if $\mathsf{dom}(\mathbf{s}') = \mathsf{dom}(\mathbf{s})$ and $\mathbf{s}'(\alpha) \leqslant \mathbf{s}(\alpha)$ for every $\alpha \in \mathsf{dom}(\mathbf{s})$. We say that a type map is *usable* if so is each type in its image. We let Φ range over *constraint sets* and write $\mathsf{tv}(\Phi)$ for the set of type variables occurring in (the constraints of) Φ. We can now define the solution of a constraint set precisely:

Definition 8 (constraint set solution)**.** We say that a type map $\mathbf{s}$ is a *solution* of Φ if:

1. $\mathsf{tv}(\Phi) \subseteq \mathsf{dom}(\mathbf{s})$, and
2. $\bot \notin \Phi$, and
3. for all $\omega \leqslant \tau \in \Phi$ we have $\mathbf{s}(\omega) \leqslant \mathbf{s}(\tau)$, and
4. for all $\mathcal{S} \sqsubset \tau \in \Phi$ and $\mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\tau)$ there exists $\mathtt{m}(\overline{\sigma}) \in \mathcal{S}$ such that $|\overline{\sigma}| = |\overline{\omega}|$, and
5. for all $\mathcal{S} \sqsubset \tau \in \Phi$, $\mathtt{m}(\overline{\sigma}) \in \mathcal{S}$, $\mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\tau)$, $1 \leq i \leq |\overline{\sigma}| = |\overline{\omega}|$ we have $\mathbf{s}(\omega_i) \leqslant \mathbf{s}(\sigma_i)$.

We now describe the generation rules in Table 3. In general they correspond closely to the typing rules of Table 2 so we will highlight only the differences. We use Δ instead of Γ for ranging over the type environments generated by the rules to emphasize the fact that they associate names with type expressions rather than types. We extend to generated environments the same notation introduced for plain type environments.

Judgments for processes have the form $P \blacktriangleright \Delta; \Phi$ where Δ and Φ are the type environment and constraint set generated from P. Rule [G-NULL] is unremarkable. In rule [G-SEND] all types that are unconstrained – and therefore cannot be inferred – are represented by fresh type variables. For this reason, there is no side condition that verifies their usability. We will take care of this condition later on, by restricting our interest to those solutions of constraint sets that only assign

Generation rules for processes $P \blacktriangleright \Delta; \Phi$

[G-SUB]
$$\frac{P \blacktriangleright \Delta; \Phi}{P \blacktriangleright \Delta, u : \mathbb{1}; \Phi}$$

[G-NULL]
$$\frac{}{\texttt{null} \blacktriangleright -; \emptyset}$$

[G-SEND]
$$\frac{}{u\,!\,\mathtt{m}(\overline{u}) \blacktriangleright u : \mathtt{m}(\overline{\alpha}) \cdot \prod_{1 \le i \le n} u_i : \alpha_i; \emptyset} \quad \alpha_1, \ldots, \alpha_n \text{ fresh}$$

[G-PAR]
$$\frac{P_i \blacktriangleright \Delta_i; \Phi_i \ ^{(1 \le i \le 2)}}{P_1 \ \& \ P_2 \blacktriangleright \Delta_1 \cdot \Delta_2; \Phi_1 \cup \Phi_2}$$

[G-OBJECT]
$$\frac{a : g \vdash C \blacktriangleright \Phi_1 \qquad P \blacktriangleright \Delta, a : \tau; \Phi_2}{\texttt{object}\ a : g\ [C]\ P \blacktriangleright \Delta; \Phi_1 \cup \Phi_2 \cup \{ g \preccurlyeq \tau \}}$$

Generation rules for patterns $\Delta \vdash_{\mathcal{S}} J :: \overline{\mathtt{m}}$

[G-MSG]
$$\frac{}{\overline{x : \omega} \vdash_{\mathcal{S}} \mathtt{m}(\overline{x}) :: \mathtt{m}} \quad \mathtt{m}(\overline{\omega}) \in \mathcal{S} \quad \mathsf{usable}(\overline{\omega})$$

[G-JOIN]
$$\frac{\Delta_i \vdash_{\mathcal{S}} J_i :: \overline{\mathtt{m}}_i \ ^{(1 \le i \le 2)}}{\Delta_1, \Delta_2 \vdash_{\mathcal{S}} J_1 \ \& \ J_2 :: \overline{\mathtt{m}}_1, \overline{\mathtt{m}}_2} \quad \overline{\mathtt{m}}_1 \text{ and } \overline{\mathtt{m}}_2 \text{ disjoint}$$

Generation rules for classes $a : g \vdash C \blacktriangleright \Phi$

[G-REACTION]
$$\frac{\overline{x : \omega} \vdash_{\mathsf{sig}(g)} J :: \overline{\mathtt{m}} \qquad P \blacktriangleright \overline{x : \tau}, a : \sigma; \Phi}{a : g \vdash J \rhd P \blacktriangleright \Phi \cup \{ \overline{\omega \preccurlyeq \tau}, g \preccurlyeq g[\overline{\mathtt{m}}] \cdot \sigma \}} \quad \mathsf{usable}(g[\overline{\mathtt{m}}])$$

[G-CLASS]
$$\frac{a : g \vdash C_i \blacktriangleright \Phi_i \ ^{(1 \le i \le 2)}}{a : g \vdash C_1 \mid C_2 \blacktriangleright \Phi_1 \cup \Phi_2}$$

Table 3: Generation rules.

usable types to type variables. Rule [G-PAR] is also straightforward. Note that constraint sets generated from the composed processes are merged in the conclusion. Rule [G-OBJECT] differs from [T-OBJECT] in two ways. The first difference is that we slightly generalize the syntax of processes and allow the type annotation supplied by the programmer to be a guarded type expression g instead of a type t. This allows the programmer to annotate a with a partially specified type in hopes that the type checking algorithm manages to infer the missing bits. We will see in Section 4 that there are some caveats as to when the inference is successful. The second difference is that it is not possible to compare the types g and τ right away for these type expressions will, in general, contain type variables. Consequently, the relation $g \preccurlyeq \tau$ is only symbolically recorded in the constraint set of the conclusion, along with any other constraint generated from P. Rule [G-SUB] roughly corresponds to [T-SUB], except that it allows for a limited form of subtyping between type environments, whereby the one in the conclusion is allowed to have an additional name u with associated type $\mathbb{1}$. Note that u is necessarily different from those already in Δ since the composition $\Delta, u : \mathbb{1}$ is defined only if $u \notin \mathsf{dom}(\Delta)$. Note also that this rule is structural, which makes the generation rules not algorithmic, strictly speaking. However, unlike [T-SUB], the places where it may be necessary to apply [G-SUB] are easy to spot. These are the premises of [G-OBJECT] and [G-REACTION], which make assumptions on the presence of particular names in the environment.

Judgments for patterns have the form $\Delta \vdash_{\mathcal{S}} J :: \overline{\mathtt{m}}$ and are essentially the same of Table 2. Note that here $\mathcal{S}$ may contain "incomplete" message types with type variables.

Judgments for classes have the form $a : g \vdash C \blacktriangleright \Phi$ where Φ is the constraint set that must be satisfied in order for C to be well typed. As expected, the interesting bits are all in [G-REACTION]. First of all, the signature $\mathsf{sig}(g)$ used in the judgment for the pattern is computed from a guarded

expression and not from a type. Second, we add constraints of the form $\omega_i \preccurlyeq \tau_i$ for each argument in the pattern. In [T-REACTION] the type of each argument was required to match exactly the one it had when typing P, but this was possible because of the availability of a more general subsumption rule [T-SUB]. The side condition $t \preccurlyeq t[\overline{\mathtt{m}}] \cdot s$ of [T-REACTION] becomes the constraint $g \preccurlyeq g[\overline{\mathtt{m}}] \cdot \sigma$ in the conclusion of [G-REACTION]. Note that allowing the type annotation to be a general type expression as opposed to a guarded expression would require dealing with constraints representing *differential inequations* between types.

We now formalize the relationship between the typing rules and the generation rules:

Theorem 1 (correctness and completeness of generation rules)**.** *The following properties hold:*

1. *If* $P \blacktriangleright \Delta; \Phi$ *and* $\mathbf{s}$ *is a usable solution of* Φ*, then* $\mathbf{s}(\Delta) \vdash P$*.*
2. *If* $\Gamma \vdash P$*, then there exist* Δ*,* Φ *and a usable solution* $\mathbf{s}$ *of* Φ *such that* $P \blacktriangleright \Delta; \Phi$ *and* $\Gamma \preccurlyeq \mathbf{s}(\Delta)$*.*

Proof. See Appendix A.1. □

Example 4. We apply the generation rules to Example 1, which we proved to be well typed in Example 3. To make the example slightly more interesting, instead of using t_{lock} and t_{user} we annotate *lock* and *user* with the guarded expressions

$$\begin{array}{lcl} g_{lock} & \stackrel{\text{def}}{=} & *\mathtt{Acquire}(\beta_1) \cdot (\mathtt{FREE} + \mathtt{BUSY} \cdot \mathtt{Release}) \\ g_{user} & \stackrel{\text{def}}{=} & *\mathtt{Reply}(\alpha_1) \end{array}$$

which is the minimal amount of annotation required by our type checking algorithm. To make it easier to follow the steps of the constraint generation phase, we name type variables consistently so that the α_i always refer to *lock* and the β_j always refer to *user*.

For the process in the first reaction of *lock* we obtain the derivation $\mathcal{D}_4$

$$\dfrac{\dfrac{}{lock\,!\,\mathtt{BUSY} \blacktriangleright lock : \mathtt{BUSY}; \emptyset} \quad \dfrac{}{sender\,!\,\mathtt{Reply}(lock) \blacktriangleright sender : \mathtt{Reply}(\alpha_2), lock : \alpha_2; \emptyset}}{lock\,!\,\mathtt{BUSY}\ \&\ sender\,!\,\mathtt{Reply}(lock) \blacktriangleright sender : \mathtt{Reply}(\alpha_2), lock : \mathtt{BUSY} \cdot \alpha_2; \emptyset}$$

by means of two applications of [G-SEND] and one application of [G-PAR]. At this stage we do not know how *lock* is going to be used by the receiver of the `Reply` message. Consequently, the type of *lock* is unknown and denoted by the type variable α_2, which also occurs in the type expression associated with *sender*. We can now handle the first reaction of *lock*, obtaining

$$\dfrac{\vdots \qquad \mathcal{D}_4}{lock : t_{lock} \vdash \mathtt{FREE}\ \&\ \mathtt{Acquire}(sender) \rhd lock\,!\,\mathtt{BUSY}\ \&\ sender\,!\,\mathtt{Reply}(lock) \blacktriangleright \Phi}\ [\text{G-REACTION}]$$

where $\Phi = \{\beta_1 \preccurlyeq \mathtt{Reply}(\alpha_2), g_{lock} \preccurlyeq *\mathtt{Acquire}(\beta_1) \cdot \mathtt{BUSY} \cdot \alpha_2\}$. The first constraint concerns the correct use of *sender*, whereas the second constraint imposes that the firing of the reaction moves *lock* into one of the configurations that are valid according to g_{lock}. We omit the derivation concerning the pattern which is isomorphic to $\mathcal{D}_1$ in Example 3.

The second reaction of *lock* is handled similarly by the derivation

$$\dfrac{\dfrac{}{- \vdash_{\mathcal{S}} \mathtt{BUSY}\ \&\ \mathtt{Release} :: \mathtt{BUSY} \cdot \mathtt{Release}}\ [\text{G-MSG}] \quad \dfrac{}{lock\,!\,\mathtt{FREE} \blacktriangleright lock : \mathtt{FREE}; \emptyset}\ [\text{G-SEND}]}{lock : g_{lock} \vdash \mathtt{BUSY}\ \&\ \mathtt{Release} \rhd lock\,!\,\mathtt{FREE} \blacktriangleright \{g_{lock} \preccurlyeq *\mathtt{Acquire}(\beta_1) \cdot \mathtt{FREE}\}}\ [\text{G-REACTION}]$$

Input: A constraint set Φ generated by the rules in Table 3.
Output: Either **fail** if Φ is unsatisfiable or the largest solution for Φ.

1. Compute the largest set $\Phi_{\mathbf{der}}$ of constraints that are derivable from Φ. (Section 4.1)
2. Compute the largest solution $\mathbf{s}$ of the upper bounds $\alpha \leqslant \tau \in \Phi_{\mathbf{der}}$. (Section 4.2)
3. If either $\bot \in \Phi_{\mathbf{der}}$ or $g \leqslant \tau \in \Phi_{\mathbf{der}}$ and $\mathbf{s}(g) \not\leqslant \mathbf{s}(\tau)$, then **fail**. (Section 4.3)
4. Return $\mathbf{s}$. (Section 4.4)

Figure 1: Constraint resolution algorithm.

which determines another constraint for the state reached by *lock* after it has consumed `BUSY` and `Release` messages.

Concerning the sole reaction of *user* we obtain the derivation $\mathcal{D}_5$

$$\dfrac{\dfrac{}{lock : \alpha_1 \vdash_{\mathsf{sig}(g_{user})} \texttt{Reply}(lock) :: \texttt{Reply}} \quad \dfrac{\dfrac{}{lock!\texttt{Release} \blacktriangleright lock : \texttt{Release}; \emptyset}}{lock!\texttt{Release} \blacktriangleright lock : \texttt{Release}, user : \mathbb{1}; \emptyset}}{user : g_{user} \vdash \texttt{Reply}(lock) \rhd lock!\texttt{Release} \blacktriangleright \{\, \alpha_1 \leqslant \texttt{Release}, g_{user} \leqslant g_{user} \cdot \mathbb{1} \,\}}$$

where the sub-derivation on the right makes use of [G-SUB] to introduce *user* in the type environment. Because the name *user* does not occur in the right-hand-side of the reaction, its type is $\mathbb{1}$. The constraint $g_{user} \leqslant g_{user} \cdot \mathbb{1}$ verifies that "not using *user*" is allowed according to g_{user}.

Concerning the last line of the term we first obtain the derivation $\mathcal{D}_6$

$$\dfrac{\dfrac{}{lock!\texttt{Acquire}(user) \blacktriangleright lock : \texttt{Acquire}(\beta_2), user : \beta_2; \emptyset}\,[\text{G-SEND}] \qquad \vdots}{lock!\texttt{Acquire}(user)\ \&\ \cdots \blacktriangleright lock : \texttt{Acquire}(\beta_2) \cdot \texttt{Acquire}(\beta_3), user : \beta_2 \cdot \beta_3; \emptyset}\,[\text{G-PAR}]$$

whose rightmost sub-derivation is essentially the same as the leftmost one, except that a different type variable β_3 is introduced. Both β_2 and β_3 represent the non-local (hence unknown) usages of the two occurrences of *user*. We use $\mathcal{D}_6$ in the derivation

$$\dfrac{\dfrac{}{lock!\texttt{FREE} \blacktriangleright lock : \texttt{FREE}; \emptyset}\,[\text{G-SEND}] \qquad \mathcal{D}_6}{lock!\texttt{FREE}\ \&\ \cdots \blacktriangleright lock : \texttt{FREE} \cdot \texttt{Acquire}(\beta_2) \cdot \texttt{Acquire}(\beta_3), user : \beta_2 \cdot \beta_3; \emptyset}\,[\text{G-PAR}]$$

to deal with the last line of the term. The generation phase for the whole term is completed by two subsequent applications of [G-OBJECT], which add the constraints $g_{user} \leqslant \beta_2 \cdot \beta_3$ and $g_{lock} \leqslant \texttt{FREE} \cdot \texttt{Acquire}(\beta_2) \cdot \texttt{Acquire}(\beta_3)$ to those generated so far. These constraints ensure that *lock* and *user* are *used* according to the respective types. ■

4. Constraint Resolution

In this section we tackle the problem of checking whether a given constraint set is satisfiable. If this is the case, we also want to determine its *most precise* solution. Since we are working with behavioral types, by "most precise" we mean "least permitting", that is "larger" according to $\leqslant$. Indeed, a (smaller) type that permits all behaviors is arguably less interesting than a (larger) type that permits only the behaviors that are necessary for typing a process. The algorithm that resolves a constraint set is shown in Figure 1 and each of its steps is fleshed out in its sub-section.

4.1. Constraint Derivation

In this step the algorithm computes the largest *derivable constraint set* $\Phi_{\mathbf{der}}$ starting from the constraint set Φ generated by the rules in Table 3. The idea is to make explicit all the constraints that are only implicitly contained in Φ so that it is easier to check whether Φ is satisfiable and to compute its most precise solution, provided there is one.

To illustrate constraint derivation in a simple case, consider the constraint set that contains only $\texttt{Acquire(Reply(Release))} + \mathbb{1} \leqslant \texttt{Acquire}(\alpha)$ and recall that our goal is to find out whether this constraint set is satisfiable. From the definition of subtyping (Definition 4), we know that the types of the arguments of the `Acquire` messages on the two sides of the constraint must be related contravarianly. Therefore, the above constraint is satisfiable provided that the constraint $\alpha \leqslant \texttt{Reply(Release)}$ is satisfiable as well. Notice that this latter constraint provides an upper bound for the type variable α which is also the most precise instantiation for α if we want the initial constraint set to be satisfied.

In general, $\Phi_{\mathbf{der}}$ is obtained from Φ by deriving all the constraints *implied* by those in Φ using the properties of subtyping. The judgments $\Phi \Vdash \varphi$ used to derive implicit constraints are those inferrable by the following deduction system:

$$\text{[s1]}\ \frac{}{\Phi \cup \{\varphi\} \Vdash \varphi} \qquad \text{[s2]}\ \frac{\Phi \Vdash g \leqslant \tau}{\Phi \Vdash \mathsf{sig}(g) \sqsubset \tau} \qquad \text{[s3]}\ \frac{\Phi \Vdash \mathcal{S} \sqsubset \tau \qquad \Phi \Vdash \alpha \leqslant \sigma}{\Phi \Vdash \mathcal{S} \sqsubset \sigma}\ \alpha \in \mathsf{utv}(\tau)$$

$$\text{[s4]}\ \frac{\Phi \Vdash \mathcal{S} \sqsubset \tau \quad \mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\tau) \quad \mathtt{m}(\overline{\sigma}) \in \mathcal{S} \Rightarrow |\overline{\sigma}| \neq |\overline{\omega}|}{\Phi \Vdash \bot} \qquad \text{[s5]}\ \frac{\Phi \Vdash \mathcal{S} \sqsubset \tau \quad \mathtt{m}(\sigma_1, \ldots, \sigma_n) \in \mathcal{S} \quad \mathtt{m}(\omega_1, \ldots, \omega_n) \in \mathsf{sig}(\tau) \quad 1 \leq i \leq n}{\Phi \Vdash \omega_i \leqslant \sigma_i}$$

The axiom [s1] simply states that every constraint φ in Φ is trivially derivable from Φ. When $\Phi \Vdash g \leqslant \tau$, rule [s2] derives a constraint of the form $\mathsf{sig}(g) \sqsubset \tau$ recording that we expect each message type in τ to be matched by at least a corresponding one in $\mathsf{sig}(g)$ with the same tag and arity, as by clause 2 of Definition 4. Rule [s3] propagates this information to every upper bound σ of a type variable α that occurs unguarded in τ. Rule [s4] checks clause 2 of Definition 4: if $\mathcal{S} \sqsubset \tau$ and there exists a message type $\mathtt{m}(\overline{\omega})$ in $\mathsf{sig}(\tau)$ such that no message type $\mathtt{m}(\overline{\sigma})$ in $\mathcal{S}$ has the same arity, then a fatal inconsistency has been detected in the constraint set and the unsatisfiable constraint $\bot$ is derived. Rule [s5] corresponds to clause 3 of Definition 4: whenever we find two matching signatures $\mathtt{m}(\overline{\sigma})$ and $\mathtt{m}(\overline{\omega})$ with the same tag and arity, we derive all the constraints that contravariantly relate corresponding argument types.

Two properties of $\Phi_{\mathbf{der}}$ are key in our setting. The first one is that $\Phi_{\mathbf{der}}$ includes Φ and is finite if so is Φ. Indeed, every derived constraint is made of (sets of) subterms occurring in Φ and the set of distinct subterms occurring in Φ is finite too. Therefore, $\Phi_{\mathbf{der}}$ can be effectively computed by a simple fixed-point procedure starting from Φ. The second property relates the solutions of Φ with those of $\Phi_{\mathbf{der}}$. In order to formalize this relationship, we must introduce some terminology for comparing constraint sets that will be useful also in the following:

Definition 9 (constraint set equivalence)**.** We say that Φ_1 and Φ_2 are *strongly equivalent* if they have the same solutions and that they are *weakly equivalent* if they have the same largest solution.

Lemma 1. *The constraint sets Φ and $\Phi_{\mathbf{der}}$ are strongly equivalent.*

Proof. It is enough to show that, if $\Phi \Vdash \varphi$, then every solution $\mathbf{s}$ of Φ is also a solution of $\{\varphi\}$. This follows from a simple induction on the derivation of $\Phi \Vdash \varphi$ using the definition of $\leqslant$. □

Example 5. Below is the final constraint set generated in Example 4:

$$\begin{array}{ll} \beta_1 \preccurlyeq \texttt{Reply}(\alpha_2) & g_{lock} \preccurlyeq *\texttt{Acquire}(\beta_1) \cdot \texttt{BUSY} \cdot \alpha_2 \\ \alpha_1 \preccurlyeq \texttt{Release} & g_{lock} \preccurlyeq *\texttt{Acquire}(\beta_1) \cdot \texttt{FREE} \\ g_{user} \preccurlyeq g_{user} \cdot \mathbb{1} & g_{lock} \preccurlyeq \texttt{FREE} \cdot \texttt{Acquire}(\beta_2) \cdot \texttt{Acquire}(\beta_3) \\ g_{user} \preccurlyeq \beta_2 \cdot \beta_3 & \end{array}$$

From these constraints we derive the following signature constraints using [s2], where $\mathcal{S}_{lock} = \mathsf{sig}(g_{lock}) = \{\, \texttt{FREE}, \texttt{BUSY}, \texttt{Acquire}(\beta_1), \texttt{Release} \,\}$ and $\mathcal{S}_{user} = \mathsf{sig}(g_{user}) = \{\, \texttt{Reply}(\alpha_1) \,\}$:

$$\begin{array}{ll} \mathcal{S}_{user} \sqsubset g_{user} \cdot \mathbb{1} & \mathcal{S}_{lock} \sqsubset *\texttt{Acquire}(\beta_1) \cdot \texttt{BUSY} \cdot \alpha_2 \\ \mathcal{S}_{user} \sqsubset \beta_2 \cdot \beta_3 & \mathcal{S}_{lock} \sqsubset *\texttt{Acquire}(\beta_1) \cdot \texttt{FREE} \\ & \mathcal{S}_{lock} \sqsubset \texttt{FREE} \cdot \texttt{Acquire}(\beta_2) \cdot \texttt{Acquire}(\beta_3) \end{array}$$

From these, repeated applications of [s3] and [s5] allow us to further derive

$$\alpha_1 \preccurlyeq \alpha_1 \qquad \beta_1 \preccurlyeq \beta_1 \qquad \beta_2 \preccurlyeq \beta_1 \qquad \beta_3 \preccurlyeq \beta_1 \qquad \alpha_2 \preccurlyeq \alpha_1$$

thus completing the set of derivable constraints. ■

4.2. Largest Solution of Upper Bounds

In this section we describe a procedure for computing the largest solution of an arbitrary set of upper bounds $\Phi_{\mathbf{up}} = \{\, \alpha_i \preccurlyeq \tau_i \,\}_{i \in I}$. In the constraint resolution algorithm (Figure 1) this procedure is applied to the set of upper bounds that are found in $\Phi_{\mathbf{der}}$. The procedure updates $\Phi_{\mathbf{up}}$ through a number of sub-steps detailed below. For each sub-step we show that the updated constraint set is weakly/strongly equivalent to the previous one.

Unbounded Type Variables. First of all we check whether $\mathsf{tv}(\Phi) = \{\, \alpha_i \,\}_{i \in I}$. If this is not the case, we add vacuous upper bounds $\alpha \preccurlyeq \mathbb{0}$ to $\Phi_{\mathbf{up}}$ for the variables α not having one in the set, being $\mathbb{0}$ the largest type. The obtained constraint set is trivially strongly equivalent to the original one and is guaranteed to contain at least one upper bound for each type variable occurring in Φ.

Grouping Constraints. We group those constraints concerning the same type variable using the property that the type $t + s$ is the greatest lower bound of t and s. More precisely, we rewrite $\Phi_{\mathbf{up}}$ as $\{\, \alpha_i \preccurlyeq \sum_{j \in I, \alpha_i = \alpha_j} \tau_j \,\}_{i \in I}$. Again, it is a simple exercise to show that the obtained constraint set is strongly equivalent to the previous one. From now on we may assume that $\Phi_{\mathbf{up}}$ contains *exactly* one upper bound for each type variable occurring in Φ.

Guarded Upper Bounds. This sub-step rewrites $\Phi_{\mathbf{up}}$ so that each constraint has a guarded expression on the right hand side to obtain a constraint set of the form $\{\, \alpha_i \preccurlyeq g_i \,\}_{i \in I}$. In general, it is not possible to perform such rewriting while guaranteeing that the obtained constraint set is strongly equivalent to the original one, but given that we are interested in finding the largest solution of $\Phi_{\mathbf{up}}$ the weak form of equivalence suffices.

The main difficulty in rewriting a constraint $\alpha \preccurlyeq \tau$ into a weakly equivalent one $\alpha \preccurlyeq g$ is that the type variable α may occur unguarded and non-linearly in τ. Hopkins and Kozen [24] have shown that the unique largest solution of this constraint, which we denote by $\mathsf{HK}(\alpha, \tau)$, can be computed by means of purely symbolic manipulations of τ. Below we just recall the key definitions and results that are useful for our purposes, referring the interested reader to Hopkins and Kozen [24] for more details. Hereafter, we write $\tau\{\!\{\sigma/\alpha\}\!\}$ for the type expression obtained by replacing all and only the *unguarded* occurrences of α in τ with σ.

Definition 10 (Hopkins-Kozen solution)**.** Let the *differential* of a type expression τ with respect to a type variable α, denoted by $\tau[\alpha]$, be the unique differential operator such that $\alpha[\alpha] = \mathbb{1}$ and $\beta[\alpha] = \mathbb{0}$ if $\alpha \neq \beta$ and $\mathtt{M}[\alpha] = \mathbb{0}$. We define the Hopkins-Kozen solution of $\alpha \leqslant \tau$ as $\mathsf{HK}(\alpha, \tau) \stackrel{\text{def}}{=} (*\,(\tau[\alpha]\{\!\{\tau/\alpha\}\!\}) \cdot \tau)\,\{\!\{\mathbb{0}/\alpha\}\!\}$.

A couple of examples will help understanding Definition 10. Consider first the upper bound $\alpha \leqslant \tau$ where $\tau \stackrel{\text{def}}{=} \mathtt{A} \cdot \alpha + \mathtt{B}$. In this case we have $\tau[\alpha] \simeq \mathtt{A}$ and therefore $\mathsf{HK}(\alpha, \tau) \simeq *\mathtt{A} \cdot \mathtt{B}$, which coincides with the well-known (least) solution of the equation $\alpha = \tau$ when τ is interpreted as a conventional (*i.e.*, non-commutative) regular expression. Suppose instead that $\tau \stackrel{\text{def}}{=} \mathtt{A} \cdot \alpha \cdot \alpha + \mathtt{B}$ where α occurs non-linearly. In this case $\tau[\alpha] \simeq \mathtt{A} \cdot \alpha$ hence $\mathsf{HK}(\alpha, \tau) \simeq *(\mathtt{A} \cdot \mathtt{B}) \cdot \mathtt{B}$.

Notice that $\alpha \notin \mathsf{utv}(\mathsf{HK}(\alpha, \tau))$ and that α may still occur in $\mathsf{HK}(\alpha, \tau)$, but only within a message argument type. The next Theorem, which hinges on results proved by Hopkins and Kozen, states that $\mathsf{HK}(\alpha, \tau)$ does indeed represent the largest solution of the given constraint regardless of how other type variables possibly occurring in τ are substituted.

Theorem 2. *Let* $\mathbf{s}$ *be any type map such that* $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\tau) \setminus \{\,\alpha\,\}$ *and* s *be the unique regular tree that satisfies the equation* $\alpha = \mathbf{s}(\mathsf{HK}(\alpha, \tau))$*. Then* s *is the unique largest solution of* $\alpha \leqslant \tau$*. In particular,* $s \leqslant \mathbf{s}(\tau)\{s/\alpha\}$ *and if* $t \leqslant \mathbf{s}(\tau)\{t/\alpha\}$*, then* $t \leqslant s$*.*

Proof. Both the property $[\![\mathbf{s}(\tau)\{s/\alpha\}]\!] \subseteq [\![s]\!]$ and the property $[\![\mathbf{s}(\tau)\{t/\alpha\}]\!] \subseteq [\![t]\!]$ implies $[\![s]\!] \subseteq [\![t]\!]$ follow directly from Hopkins and Kozen [24, Theorem 4.1]. We can conclude the proof if we show that $\mathsf{sig}(\mathbf{s}(\tau)\{s/\alpha\}) \subseteq \mathsf{sig}(s)$. We reason on the three possibilities by which a message type $\mathtt{m}(\bar{t})$ may end up in $\mathsf{sig}(\mathbf{s}(\tau)\{s/\alpha\})$. It may be the case that $\mathtt{m}(\bar{t}) = \mathbf{s}(\mathtt{m}(\overline{\omega}))$ and $\mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\tau)$. Then $\mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\mathsf{HK}(\alpha, \tau))$ hence $\mathtt{m}(\bar{t}) \in \mathsf{sig}(s)$. It may be the case that $\mathtt{m}(\bar{t}) \in \mathsf{sig}(\mathbf{s}(\beta))$ for some $\beta \in \mathsf{utv}(\tau) \setminus \{\,\alpha\,\}$. Then $\beta \in \mathsf{utv}(\mathsf{HK}(\alpha, \tau))$ hence $\mathtt{m}(\bar{t}) \in \mathsf{sig}(s)$. Finally, it may be the case that $\mathtt{m}(\bar{t}) \in \mathsf{sig}(s)$ and $\alpha \in \mathsf{utv}(\tau)$, but then there is nothing to prove. □

Corollary 1. *The constraints* $\alpha \leqslant \tau$ *and* $\alpha \leqslant \mathsf{HK}(\alpha, \tau)$ *are weakly equivalent.*

Example 6. It is not always the case that the constraints $\alpha \leqslant \tau$ and $\alpha \leqslant \mathsf{HK}(\alpha, \tau)$ are strongly equivalent. To see why, consider $\tau = \mathsf{a} \cdot \alpha + \mathsf{b}$. We have $\tau[\alpha] = \mathsf{a}$ and $\mathsf{HK}(\alpha, \tau) = *\mathsf{a} \cdot \mathsf{b}$. Now the type map $\{\,\alpha \mapsto *\mathsf{a} \cdot \mathsf{b} + \mathsf{c}\,\}$ is a solution of $\alpha \leqslant \mathsf{HK}(\alpha, \tau)$ but not of $\alpha \leqslant \tau$. ■

With these tools at hand we can easily come up with a procedure for rewriting an arbitrary constraint set $\{\,\alpha_i \leqslant \tau_i\,\}_{i \in I}$ into a weakly equivalent one $\{\,\alpha_i \leqslant g_i\,\}_{i \in I}$. Take a type variable α_i that occurs unguarded in some of the τ_j and compute $\sigma = \mathsf{HK}(\alpha_i, \tau_i)$. Replace the constraint $\alpha_i \leqslant \tau_i$ with $\alpha_i \leqslant \sigma$ and furthermore substitute each unguarded occurrence of α_i in the remaining constraints with σ. After these operations we are left with a weakly equivalent constraint set in which there are no more unguarded occurrences of α_i. It suffices to iterate the procedure for all the remaining type variables that do occur unguarded in some of the τ_j to complete the rewriting.

Largest Solution Synthesis. Given a constraint set of the form $\{\,\alpha_i \leqslant g_i\,\}_{i \in I}$ with one constraint for each type variable occurring in it, its largest solution $\mathbf{s}$ is uniquely determined as the tuple of regular trees satisfying the system of equations $\{\,\alpha_i = g_i\,\}_{i \in I}$. Existence and unicity of such solution are guaranteed by Courcelle [11, Theorem 4.3.1] using the fact that all g_i are guarded.

Example 7. Looking at the overall set of constraints in Example 5 we see that every type variable has at least one upper bound, and α_1 and β_2 have two. By grouping the upper bounds of each type variable we obtain the constraint set

$$\alpha_1 \leqslant \mathtt{Release} + \alpha_1 \qquad \beta_1 \leqslant \mathtt{Reply}(\alpha_2) + \beta_1 \qquad \beta_2 \leqslant \beta_1 \qquad \beta_3 \leqslant \beta_1 \qquad \alpha_2 \leqslant \alpha_1$$

which we now rewrite in guarded form following the procedure outlined above. For α_1 we have

$$\begin{aligned}
&\mathsf{HK}(\alpha_1, \mathtt{Release} + \alpha_1)\\
&\quad = \Big(*((\mathtt{Release} + \alpha_1)[\alpha_1]\{\!\{\mathtt{Release} + \alpha_1/\alpha_1\}\!\}) \cdot (\mathtt{Release} + \alpha_1)\Big)\{\!\{\mathbb{0}/\alpha_1\}\!\}\\
&\quad = (*(\mathbb{1}\{\!\{\mathtt{Release} + \alpha_1/\alpha_1\}\!\}) \cdot (\mathtt{Release} + \alpha_1))\{\!\{\mathbb{0}/\alpha_1\}\!\}\\
&\quad = (*\mathbb{1} \cdot (\mathtt{Release} + \alpha_1))\{\!\{\mathbb{0}/\alpha_1\}\!\} = \mathtt{Release}
\end{aligned}$$

where at each step we have simplified the result applying known algebraic equivalences. An analogous computation allows us to establish the guarded upper bound $\mathtt{Reply}(\alpha_2)$ for β_1. After substituting the computed upper bounds in place of all the unguarded occurrences of α_1 and β_2 in the remaining constraints we obtain the constraint set $\{\alpha_i \leqslant \mathtt{Release}\}_{1\leq i\leq 2} \cup \{\beta_j \leqslant \mathtt{Reply}(\alpha_2)\}_{1\leq j\leq 3}$ whose solution is $\{\alpha_i \mapsto \mathtt{Release}\}_{1\leq i\leq 2} \cup \{\beta_j \mapsto \mathtt{Reply(Release)}\}_{1\leq j\leq 3}$.

It could be argued that appealing to Hopkins and Kozen's method for rewriting upper bounds is an unnecessary complication. After all, we could have obtained the same results by simply ignoring the constraints $\alpha_1 \leqslant \alpha_1$ and $\beta_1 \leqslant \beta_1$, which say nothing relevant about α_1 and β_1. However, the procedure we have outlined can cope with arbitrary upper bounds. We will see another example in Section 5 where the full generality of Hopkins and Kozen's method is justified. ■

4.3. Constraint Set Satisfiability

Because the type map $\mathbf{s}$ computed in the previous step is the largest one that satisfies the upper bounds included in or derived from Φ, it follows that $\mathbf{s}$ is larger than every solution of Φ. The question now is whether $\mathbf{s}$ is also a solution for Φ. The following result formalizes two cases in which the answer is negative and proves the correctness of step 3 of the algorithm:

Lemma 2. *Let* $\mathbf{s}$ *be the largest solution of the upper bounds in* $\Phi_{\mathbf{der}}$. *If either* $\bot \in \Phi_{\mathbf{der}}$ *or* $g \leqslant \tau \in \Phi_{\mathbf{der}}$ *and* $\mathbf{s}(g) \not\leqslant \mathbf{s}(\tau)$, *then* Φ *is unsatisfiable.*

Proof. See Appendix A.2. □

This result calls for a decision procedure for $\leqslant$, which we now discuss. It should be noted that, because of the way $\Phi_{\mathbf{der}}$ is computed, checking the validity of the lower bounds in $\Phi_{\mathbf{der}}$ simply amounts to checking whether clause 1 of Definition 4 holds for each of them. Indeed, every instance of clause 3 has been derived and explicitly added to $\Phi_{\mathbf{der}}$. Formally:

Lemma 3. *Let* $\mathbf{s}$ *be any solution of the upper bounds in* $\Phi_{\mathbf{der}}$ *and assume* $\bot \notin \Phi_{\mathbf{der}}$. *Then* $g \leqslant \tau \in \Phi_{\mathbf{der}}$ *implies* $\mathbf{s}(g) \leqslant \mathbf{s}(\tau)$ *if and only if* $g \leqslant \tau \in \Phi_{\mathbf{der}}$ *implies* $[\![\mathbf{s}(\tau)]\!] \subseteq [\![\mathbf{s}(g)]\!]$.

Proof. The proof of the "only if" direction follows trivially from Definition 4. For the "if" direction, it suffices to show that $\leqslant \cup \{(\mathbf{s}(\omega), \mathbf{s}(\tau)) \mid \omega \leqslant \tau \in \Phi_{\mathbf{der}}\}$ is included in $\leqslant$. The proof follows easily from the definition of $\Phi_{\mathbf{der}}$, from Definition 4 and from the hypotheses. □

Thanks to Lemma 3 the problem of checking the subtyping relation between types with message arguments boils down to the problem of checking the inclusion relation between languages of message tags generated by terms of a Commutative Kleene Algebra. This problem is known to be decidable [26] and we will not detail a specific algorithm. In CobaltBlue the problem is solved by rephrasing it as the validity of a Presburger formula built from the two types, once they have been suitably normalized [10]. Currently, the tool can be configured to use either Microsoft's Z3 theorem prover [34] or the LASH toolset [36, 6] to check the validity of these formulas.

4.4. Interpretation of the Solution

If all the checks performed at step 3 are passed, then we have successfully computed the sought outcome of the algorithm:

Lemma 4. *Let* $\mathbf{s}$ *be the largest solution of the upper bounds in* $\Phi_{\mathbf{der}}$*. If* $\bot \notin \Phi_{\mathbf{der}}$ *and* $g \leqslant \tau \in \Phi_{\mathbf{der}}$ *implies* $\mathbf{s}(g) \leqslant \mathbf{s}(\tau)$*, then* $\mathbf{s}$ *is the largest solution of* Φ*.*

Proof. Immediate from Definition 8 and Lemma 1. □

Because the algorithm always terminates, from the above lemmas we conclude:

Theorem 3. *The resolution algorithm shown in Figure 1 is correct and complete.*

Notably, none of the above results guarantees that the solution obtained from the algorithm is usable and we know that this condition is key to guarantee that the process under analysis is well typed (Theorem 1). An unusable solution assigns some type variables to $\mathbb{0}$. Being $\mathbb{0}$ the top element according to subtyping, these type variables are effectively unbounded and correspond to entities of the program that are *never used nor discarded* in the program. In these cases there is simply not enough information for the type checker algorithm to infer what the type of the entity should be. A typical example is that of an infinite loop as in the object below

$$\texttt{object}\ c : \texttt{Loop}(\alpha)\ [\texttt{Loop}(x) \rhd c!\texttt{Loop}(x)] \cdots$$

where the programmer has left the type of x unspecified and x is simply passed around without ever being used. In this case, any non-$\mathbb{0}$ type (for example $\mathbb{1}$) could be used as a bogus upper bound for α without compromising the typability of the program. However, the choice of such upper bound would be completely arbitary. A more reasonable approach would be do leave c polymorphic in the type α of the message argument of Loop, so that α could be instantiated with any (usable) type. This approach would require a non-trivial extension of the type system with behavioral parametric polymorphism. For the time being, we formalize an effective, sufficient condition guaranteeing that the solution $\mathbf{s}$ obtained from the solver algorithm is usable.

Theorem 4. *If* Φ *is generated from a closed process whose type annotations are all completely specified and* $\mathbb{0}$*-free and the resolution algorithm yields a solution* $\mathbf{s}$ *of* Φ*, then* $\mathbf{s}$ *is usable.*

Proof sketch. It suffices to show that under the given hypotheses all type variables in Φ have a usable upper bound in $\Phi_{\mathbf{der}}$. Inspection of Table 3 reveals that every type variable occurring in Φ is introduced by rule [G-SEND] and occurs (also) in a message type of the form $\mathtt{m}(\overline{\alpha})$ associated to some identifier u. Since the process is closed, a lower bound for u of the form $t \leqslant \tau$ with $\mathtt{m}(\overline{\alpha}) \in \mathsf{sig}(\tau)$ is eventually generated by either [G-OBJECT] or [G-REACTION]. From the hypothesis that $\mathbf{s}$ is the largest solution of φ we know that $t = \mathbf{s}(t) \leqslant \mathbf{s}(\tau)$ holds. From clause 2 of Definition 4 we deduce that the message type $\mathtt{m}(\alpha_1, \ldots, \alpha_n)$ is matched by some $\mathtt{m}(t_1, \ldots, t_n) \in \mathsf{sig}(t)$. Hence, rule [S5] has added upper bounds $\alpha_i \leqslant t_i$ for all $1 \leq i \leq n$ to $\Phi_{\mathbf{der}}$ at step 1 of the resolution algorithm. Since t_i is a user-provided $\mathbb{0}$-free type annotation, it must be usable. Therefore, $\mathbf{s}(\alpha_i)$ is usable too. □

5. Example

In this section we discuss a variant of *lock* (Example 1) that generates more involved constraints. Specifically, we model a lock whose acquisition operation, here called `Try`, is non-blocking: *user* is always immediately notified by *lock* with either a `True` or a `False` message depending on whether the acquisition has been successful or not. If the acquisition fails, *user* non-deterministically decides to try once more or to quit. If the acquisition is successful, though, *user* has the obligation to `Release` the lock. The term modeling this scenario is shown below

$$
\begin{array}{l}
\texttt{object}\ \mathit{lock} : g_1\ [\ \texttt{FREE} \,\&\, \texttt{Try}(\mathit{sender}^{\beta_1}) \triangleright \mathit{lock}\texttt{!BUSY} \,\&\, \mathit{sender}\texttt{!True}(\mathit{lock}^{\alpha_1}) \\
\qquad\qquad\quad \mid \texttt{BUSY} \,\&\, \texttt{Try}(\mathit{sender}^{\beta_1}) \triangleright \mathit{lock}\texttt{!BUSY} \,\&\, \mathit{sender}\texttt{!False} \\
\qquad\qquad\quad \mid \texttt{BUSY} \,\&\, \texttt{Release} \quad\ \triangleright \mathit{lock}\texttt{!FREE}\ ] \\
\texttt{object}\ \mathit{user} : g_2\ [\ \texttt{WAIT}(\mathit{lock}^{\alpha_2}) \,\&\, \texttt{False} \quad\ \triangleright \mathit{user}\texttt{!WAIT}(\mathit{lock}^{\alpha_4}) \,\&\, \mathit{lock}\texttt{!Try}(\mathit{user}^{\beta_2}) \\
\qquad\qquad\quad \mid \texttt{WAIT}(\mathit{lock}^{\alpha_2}) \,\&\, \texttt{False} \quad\ \triangleright \texttt{null} \\
\qquad\qquad\quad \mid \texttt{WAIT}(\mathit{lock}^{\alpha_2}) \,\&\, \texttt{True}(l^{\alpha_3}) \triangleright l\texttt{!Release}\ ] \\
\mathit{lock}\texttt{!FREE} \,\&\, \mathit{user}\texttt{!WAIT}(\mathit{lock}^{\alpha_5}) \,\&\, \mathit{lock}\texttt{!Try}(\mathit{user}^{\beta_3})
\end{array}
$$

where $g_1 \stackrel{\text{def}}{=} *\texttt{Try}(\beta_1) \cdot (\texttt{FREE} + \texttt{BUSY} \cdot \texttt{Release})$ and $g_2 \stackrel{\text{def}}{=} \texttt{WAIT}(\alpha_2) \cdot (\texttt{False} + \texttt{True}(\alpha_3)) + \mathbb{1}$.

In *lock* there are two reactions for the `Try` operation, one for each state in which *lock* can be, and *sender* is notified accordingly. The *user* of the lock now has a state message `WAIT` carrying a reference to the lock to be acquired. There are two reactions with overlapping patterns corresponding to a `False` message from *lock*: in the first reaction *user* restores its `WAIT` message and tries to acquire the lock one more time; in the second reaction *user* terminates. As in Example 1, if *user* manages to acquire the lock, it eventually sends `Release` to it.

We do not detail the cumbersome derivations that generate the constraints. Instead, we have annotated the term with the type variables associated with message arguments and we simply report the generated constraint set:

$$
\begin{array}{llll}
\alpha_2 \leqslant \alpha_4 \cdot \texttt{Try}(\beta_2) & \beta_1 \leqslant \texttt{True}(\alpha_1) & g_1 \leqslant *\texttt{Try}(\beta_1) \cdot \texttt{BUSY} \cdot \alpha_1 & g_2 \leqslant \texttt{WAIT}(\alpha_4) \cdot \beta_2 \\
\alpha_2 \leqslant \mathbb{1} & \beta_1 \leqslant \texttt{False} & g_1 \leqslant *\texttt{Try}(\beta_1) \cdot \texttt{BUSY} & g_2 \leqslant \texttt{WAIT}(\alpha_2) \cdot \mathbb{1} \\
\alpha_3 \leqslant \texttt{Release} & & g_1 \leqslant *\texttt{Try}(\beta_1) \cdot \texttt{FREE} & g_2 \leqslant \texttt{WAIT}(\alpha_5) \cdot \beta_3 \\
 & & g_1 \leqslant \texttt{FREE} \cdot \alpha_5 \cdot \texttt{Try}(\beta_3) &
\end{array}
$$

From these constraint sets, repeated applications of the $[s^*]$ rules allow us to derive the following additional constraints

$$\alpha_1 \leqslant \alpha_3 \qquad \alpha_2 \leqslant \alpha_2 \qquad \alpha_4 \leqslant \alpha_2 \qquad \alpha_5 \leqslant \alpha_2 \qquad \beta_1 \leqslant \beta_1 \qquad \beta_2 \leqslant \beta_1 \qquad \beta_3 \leqslant \beta_1$$

eventually leading to the grouped upper bounds and rewritings summarized in the next table:

grouped upper bounds	rewrite α_2 and β_1	rewrite α_3	rewrite α_4
$\alpha_1 \leqslant \alpha_3$	$\cdots$	$\alpha_1 \leqslant \texttt{Release}$	$\cdots$
$\alpha_2 \leqslant \alpha_4 \cdot \texttt{Try}(\beta_2) + \mathbb{1} + \alpha_2$	$\alpha_2 \leqslant \alpha_4 \cdot \texttt{Try}(\beta_2) + \mathbb{1}$	$\cdots$	$\alpha_2 \leqslant *\texttt{Try}(\beta_2)$
$\alpha_3 \leqslant \texttt{Release}$	$\cdots$	$\cdots$	$\cdots$
$\alpha_4 \leqslant \alpha_2$	$\alpha_4 \leqslant \alpha_4 \cdot \texttt{Try}(\beta_2) + \mathbb{1}$	$\cdots$	$\alpha_4 \leqslant *\texttt{Try}(\beta_2)$
$\alpha_5 \leqslant \alpha_2$	$\alpha_5 \leqslant \alpha_4 \cdot \texttt{Try}(\beta_2) + \mathbb{1}$	$\cdots$	$\alpha_5 \leqslant *\texttt{Try}(\beta_2)$
$\beta_1 \leqslant \texttt{True}(\alpha_1) + \texttt{False} + \beta_1$	$\beta_1 \leqslant \texttt{True}(\alpha_1) + \texttt{False}$	$\cdots$	$\cdots$
$\beta_2 \leqslant \beta_1$	$\beta_1 \leqslant \texttt{True}(\alpha_1) + \texttt{False}$	$\cdots$	$\cdots$
$\beta_3 \leqslant \beta_1$	$\beta_1 \leqslant \texttt{True}(\alpha_1) + \texttt{False}$	$\cdots$	$\cdots$

The final rewriting step is the most interesting, since it involves resolving a constraint of the form $\alpha_4 \preccurlyeq \alpha_4 \cdot \mathtt{Try}(\beta_2) + \mathbb{1}$ where the type variable α_4 is multiplicatively combined with a non-trivial type expression in the right-hand side. In this case, Hopkins and Kozen's method yields the guarded type expression $*\mathtt{Try}(\beta_2)$.

Overall we conclude that the term is well typed and the full types of *lock* and *user* are

$$*\mathtt{Try}(\mathtt{True}(\mathtt{Release}) + \mathtt{False}) \cdot (\mathtt{FREE} + \mathtt{BUSY} \cdot \mathtt{Release})$$
$$\mathtt{WAIT}(*\mathtt{Try}(\mathtt{True}(\mathtt{Release}) + \mathtt{False})) \cdot (\mathtt{False} + \mathtt{True}(\mathtt{Release})) + \mathbb{1}$$

6. Related Work

The structural and computational complexity of type checking algorithms for concurrent objects varies widely depending on the structure of types and on the features of the model/language for which typing is defined. For example, Vasconcelos [35] defines a type system for *uniform* concurrent objects, whereby an object always exposes the same interface. In this case, object types are akin to (polymorphic) record types and the corresponding type checking algorithm can be realized following a traditional construction, whereby each occurrence of an (object) identifier is associated with a unique type. Objects with a *non-uniform* interface require a more involved type checking algorithm that usually entails a significant amount of type inference as well. This is to reconcile the fact that each occurrence of an object reference may be given a different type with the desire to minimize the amount of explicit annotations the programmer is supposed to write in the program. It would be unfeasible to require that each occurrence of an object reference is explicitly annotated with its type.

Colaço et al. [9] define a type system for non-uniform concurrent objects in which types are object interfaces listing the messages that can be sent to an object. Each message is decorated with a multiplicity that indicates how many instances of that message can be sent, with the guarantee that the object *may* move into a state that allows reception of that message. The corresponding type checking algorithm has roughly the same structure as our own, with a constraint generation phase followed by a constraint resolution phase. However, types are simpler in structure compared to those we consider, hence constraints are simpler to deal with (they essentially represent inequalities between multisets). This simpler structure of types also has consequences on the precision of the analysis. For example, according to the type system of Colaço et al. [9] it is allowed to send a `Release` message to a free lock because the lock *may* handle that message later on, after it has been acquired.

Puntigam [33] describes a type system and a corresponding type checking algorithm for typed non-uniform objects where object types are decorated with *tokens* that represent in an abstract form the internal state of objects. Once again the type checking algorithm comprises constraint generation and resolution phases concerning object states. The flat structure of tokens allows for an efficient (*i.e.* polynomial) algorithm, although the programmer is required to provide explicit type annotations, in some cases also for conditional processes.

Kouzapas et al. [28] formalize and implement a toolchain for type checking objects with structured protocols described as session types. The toolchain is able to analyze a practically relevant subset of Java code, but the choice of session types as protocol description language implies that the analysis is based on the fundamental assumption that objects with structured protocols are used linearly. As a consequence, the toolchain is unfit to analyze concurrent objects. Type checking/inference of linear resources described by session types is in general simpler and,

given a sufficiently expressive host language, it can be embedded into an existing type system without the need for a dedicated algorithm [3, 31].

A key ingredient of our type checking algorithm is the Hopkins-Kozen formula for computing the largest solution of a type constraint (Section 4.2). This formula has been shown to be a particular instance of Newtonian program analysis [14], a generic technique for solving dataflow equations that appeals to Newton's method for finding a zero of a differentiable function. In this paper, we have paired the Hopkins-Kozen formula with the known results of Courcelle [11] on finite systems of equations to extend the approach to a higher-order language.

Padovani [32] describes a refinement of the presented type system that, in addition to protocol conformance, ensures deadlock freedom as well. The additional technical machinery tracks causal dependencies between concurrent objects and is unrelated from types, therefore its implementation does not affect the type checking algorithm presented in this paper.

There is a vast literature on calculi of concurrent objects with non-uniform interfaces (Ancona et al. [3] and Hüttel et al. [25] provide extensive surveys on these works). Among them, Crafa and Padovani [12] argue that the Objective Join Calculus [16] is particularly well suited as a formal model for TSOP in a concurrent setting for two main reasons: first, the explicit representation of states and of state transitions is one of the characterizing features of TSOP [2, 17]; second, the synchronization mechanism based on join patterns [15] enables the specification of which combinations of states and operations trigger object behaviors. The same mechanism also allows the specification of compound object states. The typing discipline introduced by Crafa and Padovani [12] appears to be applicable to other models of concurrent objects. As a notable example, de'Liguoro and Padovani [13] show that a simple generalization of that typing discipline can be applied to a model of actors with first-class mailboxes. It is therefore possible that the same discipline and the type checking algorithm described in this paper might be applicable to other calculi as well, such as TyCO [35] or Mungo [28].

7. Concluding Remarks

We have demonstrated the realizability of the typing discipline of Crafa and Padovani [12] by designing a correct and complete type checking algorithm which performs a substantial amount of type inference as well. There are several aspects that deserve further investigation, especially in prospect of applying the typing discipline to the analysis of real-world programs. Below we discuss a few of them that seem to be most relevant in practice.

We have not pursued a formal analysis on the complexity of the type checking algorithm, but an inspection of its key steps allows us to draw a few preliminary conclusions. The cost of constraint generation (Section 3) is linear with respect to the size of the program being analyzed and therefore poses no particular issues. Concerning constraint resolution, we observe that each application of the Hopkins-Kozen formula has a cost that is proportional to the size of the upper bound to which it is applied, whereas the two most critical phases that appear to be potentially expensive are constraint derivation (Section 4.1) and constraint satisfaction (Section 4.3). In the current version of CobaltBlue [30], constraint derivation is implemented following the naïve fixpoint procedure suggested in Section 4.1. Experience gained while using the tool has shown that the cost of this phase is negligible when analyzing toy examples (the most complex example included in CobaltBlue is an 80 lines master-worker approximator of π), but the behavior of this phase at larger scales should be investigated in greater detail. Constraint satisfaction requires deciding the inclusion relation between expressions of a Commutative Kleene Algebra. This problem is known to be coNEXP-complete [18] and we have indeed experienced a

noticeable degradation of performances when dealing with all-permitting protocols having lots of exponentials $*$. Fortunately, these protocols are usually associated with objects that do not place any restriction on the ways they can be used. This observation prompted us to introduce in CobaltBlue the category of "objects without protocol" which can be used without limitations and are simply ignored by the type checker.

Concerning the applicability of our analysis technique to mainstream programming languages, we are encouraged by the fact that several session type disciplines have been successfully embedded in existing type systems without requiring invasive language extensions [3, 31, 27]. However, the typing discipline of Crafa and Padovani appears to be more difficult to embed in its full generality due to the presence of exotic type connectives that are loosely coupled with the syntactic structure of programs. A preliminary but promising implementation of Crafa and Padovani's typing discipline that follows the structure of the algorithm described here has been realized as a plugin for the Scala compiler to analyze Akka Actor protocols.[1]

Concerning the expressiveness of the typing discipline itself and its possible extensions, two of them seem to be particularly important. One is the already mentioned support for behavioral parametric polymorphism (Section 4.4), which could improve the modularity of the type checker and mitigate some of the aforementioned efficiency problems by reducing the size of constraint sets. The second extension aims at providing built-in support for sequential composition in protocols, which might be useful for specifying synchronous method invocations. In this respect, observe that Commutative Kleene Algebra is in fact a restriction of Concurrent Kleene Algebra [21], which provides connectives for both concurrent *and* sequential composition. Whether and how our type checking algorithm extends to a type language based on Concurrent Kleene Algebra is an open problem.

Acknowledgments. The author is grateful to Silvia Crafa for discussions on the type checking algorithm and feedback on its implementation. The author is also grateful to the anonymous reviewers for their insightful comments and suggestions of comparison with related work.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, pages 1015–1022. ACM, 2009. doi: 10.1145/1639950.1640073.

[3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3:95–230, 2016. doi: 10.1561/2500000031.

[4] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010. doi: 10.1145/1810891.1810910.

[5] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. *Theoretical Compututer Science*, 96(1):217–248, 1992. doi: 10.1016/0304-3975(92)90185-I.

[6] Bernard Boigelot. The Liège Automata-based Symbolic Handler (LASH), August 2017. URL http://www.montefiore.ulg.ac.be/~boigelot/research/lash/.

[7] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of ACM*, 11(4):481–494, 1964. doi: 10.1145/321239.321249.

[8] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On Global Types and Multi-Party Sessions. *Logical Methods in Computer Science*, 8:1–45, 2012. doi: 10.2168/LMCS-8(1:24)2012.

[1] Silvia Crafa, personal communication, 26 July 2017.

[9] Jean-Luis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in actors. In *Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, volume 139 of *IFIP Conference Proceedings*. Kluwer, 1999.

[10] John Conway. *Regular Algebra and Finite Machines*. William Clowes & Sons Ltd, 1971.

[11] Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983. doi: 10.1016/0304-3975(83)90059-2.

[12] Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 39:13:1–13:45, 2017. doi: 10.1145/3064849.

[13] Ugo de'Liguoro and Luca Padovani. Mailbox Types for Unordered Interactions. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP'18)*, LIPIcs 109. Schloss Dagstuhl, 2018.

[14] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM*, 57(6): 33:1–33:47, November 2010. ISSN 0004-5411. doi: 10.1145/1857914.1857917.

[15] Cédric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, 1996. doi: 10.1145/237721.237805.

[16] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the Join Calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003. doi: 10.1016/S1567-8326(03)00040-7.

[17] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12, 2014. doi: 10.1145/2629609.

[18] Christoph Haase and Piotr Hofman. Tightening the Complexity of Equivalence Problems for Commutative Grammars. In *Proceedings of STACS'16*, LIPIcs 47, pages 41:1–41:14, 2016. doi: 10.4230/LIPIcs.STACS.2016.41.

[19] Philipp Haller and Frank Sommers. *Actors in Scala - concurrent programming for the multi-core era*. Artima, 2011.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI'73*, pages 235–245. William Kaufmann, 1973.

[21] C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra. In *Proceedings of CONCUR'09*, LNCS 5710, pages 399–414. Springer, 2009. doi: 10.1007/978-3-642-04081-8_27.

[22] Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, volume LNCS 715, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35.

[23] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of ACM*, 63(1):9, 2016. doi: 10.1145/2827695.

[24] Mark W. Hopkins and Dexter Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proceedings of LICS'99*, pages 394–401. IEEE, 1999. doi: 10.1109/LICS.1999.782634.

[25] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016. doi: 10.1145/2873052.

[26] Dung T. Huynh. The Complexity of Equivalence Problems for Commutative Grammars. *Information and Control*, 66(1/2):103–121, 1985. doi: 10.1016/S0019-9958(85)80015-2.

[27] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-Ocaml: A Session-Based Library with Polarities and Lenses. In *Proceedings of COORDINATION'17*, LNCS 10319, pages 99–118. Springer, 2017. doi: 10.1007/978-3-319-59746-1_6.

[28] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Science of Computer Programming*, 155:52 – 75, 2018. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2017.10.006.

[29] Luca Padovani. On Projecting Processes into Session Types. *Mathematical Structures in Computer Science*, 22: 237–289, 2012. doi: 10.1017/S0960129511000405.

[30] Luca Padovani. CobaltBlue – Behavioral Type Checking for Concurrent Objects, August 2017. URL http://www.di.unito.it/~padovani/Software/CobaltBlue/index.html.

[31] Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27, 2017. doi: 10.1017/S0956796816000289.

[32] Luca Padovani. Deadlock-free typestate-oriented programming. *Programming Journal*, 2(3), 2018.

[33] Franz Puntigam. State inference for dynamically changing interfaces. *Computer Languages*, 27(4):163–202, 2001. doi: 10.1016/S0096-0551(01)00019-4.

[34] Microsoft Research. The Z3 Theorem Prover, August 2017. URL http://rise4fun.com/z3.

[35] Vasco Thudichum Vasconcelos. Typed concurrent objects. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, LNCS 821, pages 100–117. Springer, 1994. doi: 10.1007/BFb0052178.

[36] Pierre Wolper and Bernard Boigelot. On the Construction of Automata from Linear Arithmetic Constraints. In *Proceedings of TACAS'00*, LNCS 1785, pages 1–19. Springer, 2000. doi: 10.1007/3-540-46419-0_1.

Appendix A. Proofs

Appendix A.1. Proof of Theorem 1

Theorem 1 is an immediate consequence of the following two lemmas.

Lemma 5. *The following properties hold:*

1. *If* $P \blacktriangleright \Delta; \Phi$ *and* $\mathbf{s}$ *is a usable solution of* Φ*, then* $\mathbf{s}(\Delta) \vdash P$.
2. *If* $\Delta \vdash_{\mathcal{S}} J :: \overline{\mathtt{m}}$ *and* $\mathsf{tv}(\mathcal{S}) \subseteq \mathsf{dom}(\mathbf{s})$ *and* $\mathbf{s}$ *is usable, then* $\mathbf{s}(\Delta) \vdash_{\mathbf{s}(\mathcal{S})} J :: \overline{\mathtt{m}}$.
3. *If* $a : g \vdash C \blacktriangleright \Phi$ *and* $\mathbf{s}$ *is a usable solution of* Φ*, then* $a : \mathbf{s}(g) \vdash C$.

Proof. We prove all items simultaneously, by induction on the derivation of their first hypothesis and by cases on the last rule applied. Concerning the generation rules for processes, we have:

[G-SUB] Then $\Delta = \Delta', u : \mathbb{1}$ and $P \blacktriangleright \Delta'; \Phi$. By induction hypothesis we derive $\mathbf{s}(\Delta') \vdash P$. We conclude with one application of [T-SUB] by observing that $\mathbf{s}(\Delta) = \mathbf{s}(\Delta'), u : \mathbb{1} \leqslant \mathbf{s}(\Delta')$.

[G-NULL] Then $P = \mathtt{null}$ and $\Delta = -$ and $\Phi = \emptyset$. We conclude with one application of [T-NULL].

[G-SEND] Then $P = u!\mathtt{m}(\overline{u})$ and $\Delta = u : \mathtt{m}(\overline{\alpha}) \cdot \prod_{1 \leq i \leq n} u_i : \alpha_i$ and $\Phi = \emptyset$. We conclude with one application of [T-SEND] using the hypothesis that $\mathbf{s}$ is usable.

[G-PAR] Then $P = P_1 \ \& \ P_2$ and $P_i \blacktriangleright \Delta_i; \Phi_i$ for $1 \leq i \leq 2$ and $\Delta = \Delta_1 \cdot \Delta_2$ and $\Phi = \Phi_1 \cup \Phi_2$. Since $\mathbf{s}$ is a usable solution of Φ, then $\mathbf{s}$ is a usable solution also of Φ_i for $1 \leq i \leq 2$. By induction hypothesis we deduce $\mathbf{s}(\Delta_i) \vdash P_i$ for $1 \leq i \leq 2$. We conclude with one application of [T-PAR] by observing that $\mathbf{s}(\Delta) = \mathbf{s}(\Delta_1 \cdot \Delta_2) = \mathbf{s}(\Delta_1) \cdot \mathbf{s}(\Delta_2)$.

[G-OBJECT] Then $P = \mathtt{object}\ a : g\ [C]\ Q$ and $a : g \vdash C \blacktriangleright \Phi_1$ and $Q \blacktriangleright \Delta, a : \tau; \Phi_2$ and $\Phi = \Phi_1 \cup \Phi_2 \cup \{ g \preccurlyeq \tau \}$. Since $\mathbf{s}$ is a usable solution of Φ, then $\mathbf{s}$ is a usable solution also of Φ_1 and Φ_2 and furthermore $\mathbf{s}(g) \leqslant \mathbf{s}(\tau)$. By induction hypothesis we deduce $a : \mathbf{s}(g) \vdash C$ and $\mathbf{s}(\Delta), a : \mathbf{s}(\tau) \vdash Q$. We conclude with one application of [T-SUB] and one application of [T-OBJECT].

Concerning the generation rules for patterns, we have:

[G-MSG] Then $\Delta = \overline{x : \omega}$ and $J = \mathtt{m}(\overline{x})$ and $\overline{\mathtt{m}} = \mathtt{m}$ and $\mathtt{m}(\overline{\omega}) \in \mathcal{S}$ and $\mathsf{usable}(\overline{\omega})$. From $\mathsf{usable}(\overline{\omega})$ and the hypotheses that $\mathsf{tv}(\mathcal{S}) \subseteq \mathsf{dom}(\mathbf{s})$ and $\mathbf{s}$ is usable we deduce $\mathsf{usable}(\mathbf{s}(\overline{\omega}))$. From $\mathtt{m}(\overline{\omega}) \in \mathcal{S}$ we deduce $\mathtt{m}(\mathbf{s}(\overline{\omega})) \in \mathbf{s}(\mathcal{S})$. We conclude with one application of [T-MSG].

[G-JOIN] Then $\Delta = \Delta_1, \Delta_2$ and $J = J_1 \ \& \ J_2$ and $\overline{\mathtt{m}} = \overline{\mathtt{m}}_1, \overline{\mathtt{m}}_2$ and $\Delta_i \vdash_{\mathcal{S}} J_i :: \overline{\mathtt{m}}_i$ for every $1 \leq i \leq 2$ where $\overline{\mathtt{m}}_1$ and $\overline{\mathtt{m}}_2$ are disjoint. By induction hypothesis we deduce $\mathbf{s}(\Delta_i) \vdash_{\mathbf{s}(\mathcal{S})} J_i :: \overline{\mathtt{m}}_i$ for every $1 \leq i \leq 2$. We conclude with one application of [T-JOIN].

Concerning the generation rules for classes, we have:

[G-REACTION] Then $C = J \rhd P$ and $\overline{x : \omega} \vdash_{\mathsf{sig}(g)} J :: \overline{\mathtt{m}}$ and $P \blacktriangleright \overline{x : \tau}, a : \sigma; \Phi'$ and $\Phi = \Phi' \cup \{ \overline{\omega \preccurlyeq \tau}, g \preccurlyeq g[\overline{\mathtt{m}}] \cdot \sigma \}$ and $\mathsf{usable}(g[\overline{\mathtt{m}}])$. Since $\mathbf{s}$ is a usable solution of Φ, then $\mathbf{s}$ is also a usable solution of Φ' and furthermore $\overline{\mathbf{s}(\omega) \leqslant \mathbf{s}(\tau)}$ and $\overline{\mathbf{s}(g) \leqslant \mathbf{s}(g[\overline{\mathtt{m}}] \cdot \sigma) = \mathbf{s}(g)[\overline{\mathtt{m}}] \cdot \mathbf{s}(\sigma)}$ and $\mathsf{usable}(\mathbf{s}(g)[\overline{\mathtt{m}}])$. By induction hypothesis we deduce $\overline{x : \mathbf{s}(\omega)} \vdash_{\mathbf{s}(\mathcal{S})} J :: \overline{\mathtt{m}}$ and $\overline{x : \mathbf{s}(\tau)}, a : \mathbf{s}(\sigma) \vdash P$. We conclude with at most $|\overline{\omega}|$ applications of [T-SUB] followed by one application of [T-REACTION].

[G-CLASS] Then $C = C_1 \mid C_2$ and $\Phi = \Phi_1 \cup \Phi_2$ and $a : g \vdash C_i \blacktriangleright \Phi_i$ for $1 \leq i \leq 2$. Since $\mathbf{s}$ is a usable solution of Φ, then $\mathbf{s}$ is a usable solution of both Φ_1 and Φ_2. By induction hypothesis we deduce $a : \mathbf{s}(g) \vdash C_i$ for $1 \leq i \leq 2$. We conclude with one application of [T-CLASS]. □

To prove the completeness of the generation rules it is convenient to introduce a strengthened form of subtyping for type environments:

Definition 11. Let $\Gamma \sqsubseteq \Gamma'$ if and only if $\Gamma \leqslant \Gamma'$ and $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$.

Lemma 6. *The following properties hold:*

1. *If* $\Gamma \vdash P$*, then there exist* Δ, Φ *and a usable solution* $\mathbf{s}$ *of* Φ *such that* $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Phi) \cup \mathsf{tv}(\Delta)$ *and* $P \blacktriangleright \Delta; \Phi$ *and* $\Gamma \sqsubseteq \mathbf{s}(\Delta)$.
2. *If* $a : t \vdash C$*, then there exist* Φ *and a usable solution* $\mathbf{s}$ *of* Φ *such that* $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Phi)$ *and* $a : t \vdash C \blacktriangleright \Phi$.

Proof. We prove all items simultaneously, by induction on the derivation of their hypothesis and by cases on the last rule applied. Concerning the typing rules for processes, we have:

[T-SUB] Then $\Gamma \leqslant \Gamma'$ and $\Gamma' \vdash P$. By induction hypothesis there exist Δ', Φ and a usable solution $\mathbf{s}$ of Φ such that $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Phi) \cup \mathsf{tv}(\Delta')$ and $P \blacktriangleright \Delta'; \Phi$ and $\Gamma' \sqsubseteq \mathbf{s}(\Delta')$. By definition of $\Gamma \leqslant \Gamma'$ we have $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$. Let $\mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Gamma') = \{u_1, \ldots, u_n\}$. We define $\Delta \stackrel{\text{def}}{=} \Delta', u_1 : \mathbb{1}, \ldots, u_n : \mathbb{1}$. We can derive $P \blacktriangleright \Delta; \Phi$ by n applications of [G-SUB]. Note that $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta)$ by definition of Δ. Also, from the hypothesis $\Gamma \leqslant \Gamma'$ we deduce that $\Gamma(u_i) \leqslant \mathbb{1}$ for every $1 \leq i \leq n$. We conclude by observing that $\mathsf{tv}(\Delta) = \mathsf{tv}(\Delta')$ and $\Gamma \sqsubseteq \Gamma', u_1 : \mathbb{1}, \ldots, u_n : \mathbb{1} \sqsubseteq \mathbf{s}(\Delta'), u_1 : \mathbb{1}, \ldots, u_n : \mathbb{1} = \mathbf{s}(\Delta)$.

[T-NULL] Then $\Gamma = -$ and $P = \texttt{null}$. We conclude by taking $\Delta \stackrel{\text{def}}{=} -$, $\Phi \stackrel{\text{def}}{=} \emptyset$ and $\mathbf{s} \stackrel{\text{def}}{=} \{\,\}$ and with one application of [G-NULL].

[T-SEND] Then $\Gamma = u : \mathtt{m}(\tilde{t}) \cdot \prod_{1 \leq i \leq n} u_i : t_i$ and $P = u!\mathtt{m}(\tilde{u})$ and $\mathsf{usable}(\tilde{t})$. Let $\alpha_1, \ldots, \alpha_n$ be n fresh type variables and $\Delta \stackrel{\text{def}}{=} u : \mathtt{m}(\tilde{\alpha}) \cdot \prod_{1 \leq i \leq n} u_i : \alpha_i$ and $\Phi \stackrel{\text{def}}{=} \emptyset$ and $\mathbf{s} \stackrel{\text{def}}{=} \{\alpha_i \mapsto t_i\}_{1 \leq i \leq n}$. Trivially $\mathbf{s}$ is a solution of Φ, which is empty, and from the hypothesis $\mathsf{usable}(\tilde{t})$ we deduce that $\mathbf{s}$ is usable. We conclude with one application of [G-SEND] by observing that $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Delta)$ and $\Gamma = \mathbf{s}(\Delta)$.

[T-PAR] Then $\Gamma = \Gamma_1 \cdot \Gamma_2$ and $P = P_1 \,\&\, P_2$ and $\Gamma_i \vdash P_i$ for $1 \leq i \leq 2$. By induction hypothesis there exist Δ_i, Φ_i and a usable solution $\mathbf{s}_i$ of Φ_i such that $\mathsf{dom}(\mathbf{s}_i) = \mathsf{tv}(\Phi_i) \cup \mathsf{tv}(\Delta_i)$ and $P_i \blacktriangleright \Delta_i; \Phi_i$ and $\Gamma_i \sqsubseteq \mathbf{s}_i(\Delta_i)$ for $1 \leq i \leq 2$. We define $\Delta \stackrel{\text{def}}{=} \Delta_1 \cdot \Delta_2$ and $\Phi \stackrel{\text{def}}{=} \Phi_1 \cup \Phi_2$ and $\mathbf{s} \stackrel{\text{def}}{=} \mathbf{s}_1 \cup \mathbf{s}_2$, using the fact that the type variables occurring in Φ_i and Δ_i are chosen fresh and therefore $\mathsf{dom}(\mathbf{s}_1) \cap \mathsf{dom}(\mathbf{s}_2) = \emptyset$. We derive $P \blacktriangleright \Delta; \Phi$ with one application of [G-PAR] and observe that $\mathbf{s}$ is a usable solution of Φ by definition of $\mathbf{s}$. To conclude we have to show that $\Gamma \sqsubseteq \mathbf{s}(\Delta)$. The only interesting case is when we consider some $u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ for which we derive $\Gamma(u) = \Gamma_1(u) \cdot \Gamma_2(u) \leqslant \mathbf{s}_1(\Delta_1(u)) \cdot \mathbf{s}_2(\Delta_2(u)) = \mathbf{s}(\Delta(u))$.

[T-OBJECT] Then $P = \texttt{object}\ a : t\ [C]\ Q$ and $a : t \vdash C$ and $\Gamma, a : t \vdash Q$. By induction hypothesis we deduce that there exist Φ_1, $\mathbf{s}_1$, Δ', Φ_2, and $\mathbf{s}_2$ such that $\mathbf{s}_i$ is a usable solution of Φ_i for $1 \leq i \leq 2$ and $\mathsf{dom}(\mathbf{s}_1) = \mathsf{tv}(\Phi_1)$ and $\mathsf{dom}(\mathbf{s}_2) = \mathsf{tv}(\Phi_2) \cup \mathsf{tv}(\Delta')$ and $a : t \vdash C \blacktriangleright \Phi_1$ and $Q \blacktriangleright \Delta'; \Phi_2$ and $\Gamma, a : t \sqsubseteq \mathbf{s}_2(\Delta')$. It must be the case that $\Delta' = \Delta, a : \tau$ for some Δ and τ where $t \leqslant \mathbf{s}_2(\tau)$. We define $\Phi \stackrel{\text{def}}{=} \Phi_1 \cup \Phi_2 \cup \{t \leqslant \tau\}$ and $\mathbf{s} \stackrel{\text{def}}{=} \mathbf{s}_1 \cup \mathbf{s}_2$, using the fact that $\mathsf{dom}(\mathbf{s}_1) \cap \mathsf{dom}(\mathbf{s}_2) = \emptyset$. We conclude by observing that $\mathbf{s}$ is a usable solution of Φ and that $\Gamma \sqsubseteq \mathbf{s}(\Delta)$.

Concerning the typing rules for classes, we have:

[T-REACTION] Then $C = J \rhd P$ and $\Gamma \vdash_{\mathcal{S}} J :: \overline{\mathtt{m}}$ and $\Gamma, a : s \vdash P$ and $t \leqslant t[\overline{\mathtt{m}}] \cdot s$ and $\mathsf{usable}(t[\overline{\mathtt{m}}])$ where $J = \mathtt{m}_1(\overline{x_1}) \,\&\, \cdots \,\&\, \mathtt{m}_n(\overline{x_n})$. We deduce that $\{\mathtt{m}_i(\tilde{t_i})\}_{1 \leq i \leq n} \subseteq \mathcal{S}$ and $\Gamma = \overline{x_1 : t_1}, \ldots, \overline{x_n : t_n}$. Notice that the judgment $\Gamma \vdash_{\mathcal{S}} J :: \overline{\mathtt{m}}$ is derivable with straightforward applications of [G-MSG] and [G-JOIN].

By induction hypothesis we deduce that there exist $\overline{\tau_1}, \ldots, \overline{\tau_n}$, σ, Φ' and a usable solution $\mathbf{s}$ of Φ' such that $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Phi') \cup \mathsf{tv}(\sigma) \cup \bigcup_{1 \leq i \leq n} \mathsf{tv}(\overline{\tau_i})$ and $P \blacktriangleright \overline{x_1 : \tau_1}, \ldots, \overline{x_n : \tau_n}, a : \sigma; \Phi'$ and $\overline{t_i \leqslant \mathbf{s}(\tau_i)}$ for every $1 \leq i \leq n$ and $s \leqslant \mathbf{s}(\sigma)$. We define $\Phi \stackrel{\text{def}}{=} \Phi' \cup \{ \overline{t_1 \leqslant \tau_1}, \ldots, \overline{t_n \leqslant \tau_n}, t \leqslant t[\overline{\mathtt{m}}] \cdot \sigma \}$. We conclude with one application of [G-REACTION] by observing that $\mathbf{s}$ is a usable solution of Φ and $\mathsf{dom}(\mathbf{s}) = \mathsf{tv}(\Phi)$.

[T-CLASS] Then $C = C_1 \mid C_2$ and $a : t \vdash C_i$ for $i = 1, 2$. By induction hypothesis we deduce that there exist Φ_1, Φ_2, $\mathbf{s}_1$ and $\mathbf{s}_2$ such that $a : t \vdash C_i \blacktriangleright \Phi_i$ and $\mathbf{s}_i$ is a usable solution of Φ_i and $\mathsf{dom}(\mathbf{s}_i) = \mathsf{tv}(\Phi_i)$ for $1 \leq i \leq 2$. We conclude with one application of [G-CLASS] by taking $\Phi \stackrel{\text{def}}{=} \Phi_1 \cup \Phi_2$ and $\mathbf{s} \stackrel{\text{def}}{=} \mathbf{s}_1 \cup \mathbf{s}_2$. □

Appendix A.2. Proof of Lemma 2

Lemma 2 is an immediate consequence of the following result.

Lemma 7. *Let* $\mathbf{s}' \leqslant \mathbf{s}$*. If*

A. $\mathsf{dom}(\mathbf{s}) \supseteq \mathsf{tv}(g) \cup \mathsf{tv}(\tau)$*, and*
B. $\bot$ *is not derivable from* $\{ g \leqslant \tau \}$*, and*
C. $\{ g \leqslant \tau \} \Vdash \alpha \leqslant \sigma$ *implies* $\mathbf{s}(\alpha) \leqslant \mathbf{s}(\sigma)$*, and*
D. $\mathbf{s}'(g) \leqslant \mathbf{s}'(\tau)$*,*

then $\mathbf{s}(g) \leqslant \mathbf{s}(\tau)$*.*

Proof. We show that

$$\mathcal{R} \stackrel{\text{def}}{=} \leqslant \cup \{ (\mathbf{s}(\tilde{g}), \mathbf{s}(\tilde{\tau})) \mid \tilde{g} \text{ and } \tilde{\tau} \text{ satisfy the conditions A–D above} \}$$

is included in $\leqslant$. Suppose $(\mathbf{s}(\tilde{g}), \mathbf{s}(\tilde{\tau})) \in \mathcal{R}$. Then $\tilde{g}$ and $\tilde{\tau}$ satisfy the conditions A–D above. We prove conditions 1–3 of Definition 4 in order.

Concerning condition 1, observe that $[\![\mathbf{s}(\tilde{g})]\!] = [\![\mathbf{s}'(\tilde{g})]\!]$ because $\tilde{g}$ is a guarded expression. We deduce $[\![\mathbf{s}(\tilde{\tau})]\!] \subseteq [\![\mathbf{s}'(\tilde{\tau})]\!] \subseteq [\![\mathbf{s}'(\tilde{g})]\!] = [\![\mathbf{s}(\tilde{g})]\!]$ using the hypothesis $\mathbf{s}' \leqslant \mathbf{s}$ and condition D, hence condition 1 is satisfied.

Concerning condition 2, take $\mathtt{m}(\overline{s}) \in \mathsf{sig}(\mathbf{s}(\tilde{\tau}))$. We have to show that there exists $\mathtt{m}(\overline{t}) \in \mathsf{sig}(\mathbf{s}(\tilde{g}))$ with $|\overline{t}| = |\overline{s}|$. We reason on how $\mathtt{m}(\overline{s})$ may have been obtained and we have two possibilities. If $\mathtt{m}(\overline{\omega}) \in \mathsf{sig}(\tilde{\tau})$ and $\mathtt{m}(\overline{s}) = \mathbf{s}(\mathtt{m}(\overline{\omega}))$, then from the hypothesis B we deduce that there exists $\mathtt{m}(\overline{\tau}) \in \mathsf{sig}(\tilde{g})$ such that $|\overline{\tau}| = |\overline{\omega}|$ and we conclude that condition 2 holds because $\mathbf{s}(\mathtt{m}(\overline{\tau})) \in \mathsf{sig}(\mathbf{s}(g))$. If $\alpha \in \mathsf{utv}(\tilde{\tau})$ and $\mathtt{m}(\overline{s}) \in \mathsf{sig}(\mathbf{s}(\alpha))$, then from $\mathbf{s}' \leqslant \mathbf{s}$ we deduce that there exists $\mathtt{m}(\overline{s'}) \in \mathsf{sig}(\mathbf{s}'(\alpha))$ such that $|\overline{s'}| = |\overline{s}|$. From the hypothesis D we deduce that there exists $\mathtt{m}(\overline{t'}) \in \mathsf{sig}(\mathbf{s}'(\tilde{g}))$ with $|\overline{t'}| = |\overline{s'}|$. Since $\tilde{g}$ is guarded, we conclude that there exists $\mathtt{m}(\overline{t}) \in \mathsf{sig}(\mathbf{s}(\tilde{g}))$ with $|\overline{t}| = |\overline{t'}| = |\overline{s'}| = |\overline{s}|$.

Concerning condition 3, take $\mathtt{m}(t_1, \ldots, t_n) \in \mathsf{sig}(\mathbf{s}(\tilde{g}))$ and $\mathtt{m}(s_1, \ldots, s_n) \in \mathsf{sig}(\mathbf{s}(\tilde{\tau}))$. We have to prove $(s_i, t_i) \in \mathcal{R}$ for every $1 \leq i \leq n$. We reason on how $\mathtt{m}(\overline{t})$ and $\mathtt{m}(\overline{s})$ may have been obtained. Since $\tilde{g}$ is a guarded expression, it must be the case that $\mathtt{m}(\tau_1, \ldots, \tau_n) \in \mathsf{sig}(\tilde{g})$ and $t_i = \mathbf{s}(\tau_i)$ for every $1 \leq i \leq n$. For $\mathtt{m}(\overline{s})$ we have two cases.

$\boxed{\mathtt{m}(\omega_1, \ldots, \omega_n) \in \mathsf{sig}(\tilde{\tau}) \text{ and } s_i = \mathbf{s}(\omega_i) \text{ for } 1 \leq i \leq n.}$ For each i we distinguish two sub-cases:

- If $\omega_i = \alpha$ for some α, then $s_i = \mathbf{s}(\omega_i) = \mathbf{s}(\alpha)$. From condition C we deduce $\mathbf{s}(\alpha) \leqslant \mathbf{s}(\tau_i)$ hence $(s_i, t_i) \in \mathcal{R}$ because $\mathcal{R}$ includes $\leqslant$ by definition.

- If $\omega_i = g'$ for some g', then $s_i = \mathbf{s}(\omega_i) = \mathbf{s}(g')$. From condition D we deduce $\mathbf{s}'(g') \leqslant \mathbf{s}'(\tau_i)$ hence $(s_i, t_i) \in \mathcal{R}$ by definition of $\mathcal{R}$.

$\boxed{\alpha \in \mathsf{utv}(\tilde{\tau}) \text{ and } \mathtt{m}(s_1, \ldots, s_n) \in \mathsf{sig}(\mathbf{s}(\alpha))}$ From the hypothesis $\mathbf{s}' \leqslant \mathbf{s}$ we deduce $\mathtt{m}(s'_1, \ldots, s'_n) \in \mathsf{sig}(\mathbf{s}'(\alpha))$ and $s_i \leqslant s'_i$ for every $1 \leq i \leq n$. From the hypothesis $\mathbf{s}'(\tilde{g}) \leqslant \mathbf{s}'(\tilde{\tau})$ we deduce $s'_i \leqslant \mathbf{s}'(\tau_i)$ for every $1 \leq i \leq n$. From the hypothesis $\mathbf{s}' \leqslant \mathbf{s}$ we deduce $\mathbf{s}'(\tau_i) \leqslant \mathbf{s}(\tau_i) = t_i$ for every $1 \leq i \leq n$. We conclude $(s_i, t_i) \in \mathcal{R}$ for every $1 \leq i \leq n$ because $\mathcal{R}$ includes $\leqslant$ by definition. □