The Chemical Approach to Typestate-Oriented Programming

The Chemical Approach to Typestate-Oriented Programming

SILVIA CRAFA, Università di Padova
LUCA PADOVANI, Università di Torino

We introduce a novel approach to typestate-oriented programming based on the chemical metaphor: state and operations on objects are molecules of messages and state transformations are chemical reactions. This approach allows us to investigate typestate in an inherently concurrent setting, whereby objects can be accessed and modified concurrently by several processes, each potentially changing only part of their state. We introduce a simple behavioral type theory to express in a uniform way both the private and the public interfaces of objects, to describe and enforce structured object protocols consisting of possibilities, prohibitions, and obligations, and to control object sharing.

CCS Concepts: • **Theory of computation** → **Process calculi**; **Type theory**; **Object oriented constructs**; **Type structures**; *Parallel computing models*; • **Computing methodologies** → **Concurrent programming languages**;

Additional Key Words and Phrases: Typestate, Concurrency, Behavioral Types, Join Calculus

ACM Reference format:
Silvia Crafa and Luca Padovani. 2017. The Chemical Approach to Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 00 (2017), 45 pages.
DOI: 0000001.0000001

1 INTRODUCTION

In the object-oriented paradigm, the *interface* of an object describes the whole set of methods supported by the object throughout its entire lifetime. However, the usage of the object is more precisely explained in terms of its *protocol* (Beckman et al. 2011), describing the legal sequences of method calls, possibly depending on the object's internal state. Typical examples of objects with structured protocols are files, iterators, and locks: a file can be read or written only after it has been opened; an iterator can be asked to access the next element of a collection only if such element has been verified to exist; a lock should be released if (and only if) it was previously acquired. Usually, such constraints on the legal sequences of method calls are only informally documented as *comments* along with method descriptions; in this form, however, they cannot be used by the compiler to detect protocol violations.

DeLine and Fähndrich (2004) have adapted the concept of *typestate* (Strom and Yemini 1986), originally introduced for imperative programs, to the object-oriented paradigm. Typestates are machine-understandable abstractions of an object's internal state that can be used (1) to identify the subset of fields and operations that are valid when the object is in some given state and (2) to specify the effect of such operations on the state itself. For example, on a file in state `CLOSED` the compiler would permit invocations of the `open` method and forbid invocations of the `read` method, whereas on a file in state `OPEN` it would only permit invocations of `read`, `write`, and `close` methods and forbid `open`.

The first author has been supported by the University of Padova under the PRAT projects BECOM and ANCORE. The second author has been supported by ICT COST Action IC1201 BETTY, MIUR PRIN CINA, Ateneo/CSP Project SALT, and RS13MO12 DART.
Author's addresses: Silvia Crafa, Università di Padova, Dipartimento di Matematica, Italy; Luca Padovani, Università di Torino, Dipartimento di Informatica, Italy.
2017. Manuscript submitted to ACM

Furthermore, the type of open would be refined so as to specify that its invocation changes the state of the file from CLOSED to OPEN.

Typestate-oriented programming (TSOP for short) (Aldrich et al. 2009; Garcia et al. 2014; Sunshine et al. 2011) goes one step further and promotes typestates to a native feature of the programming language that encourages programmers to design objects around their protocol. Languages supporting TSOP provide explicit constructs for defining state-dependent object interfaces and implementations, for changing and possibly querying at runtime an object's typestate, and for annotating the signature of methods so as to describe their effect on the state of an object (Aldrich et al. 2009). In order to track the points in the code where the state of an object changes, hence to detect – *at compile time* – potential violations of an object's protocol using typestate information, references to objects with structured protocols are required to be stored and shared in controlled ways. Not surprisingly, then, all languages supporting static typestate checking rely on more or less sophisticated forms of aliasing control (Bierhoff and Aldrich 2007) which may hinder the applicability of typestate to objects simultaneously accessed/modified by concurrent processes. Damiani et al. (2008) have proposed an approach to conjugate typestate and concurrency in a Java-like language relying on some runtime support: users of an object can invoke any method at any time, even when the state of the object is uncertain; a method invocation is suspended until the object is in a state for which that method is legal; typestate information is used *within* methods, to make sure that only valid fields are accessed. This approach has both computational and methodological costs: it requires all methods of a concurrent object to be synchronized, it limits parallelism by sequentializing all concurrent accesses to the same object, and it guarantees protocol compliance only within methods, where some form of aliasing control can be used.

The first contribution of this paper is a foundational study of TSOP in an *inherently concurrent* setting, whereby objects can be shared and accessed concurrently, and (portions of) their state can be changed while they are simultaneously used by several processes. We base our study on the Objective Join Calculus (Fournet and Gonthier 1996, 2000) and we show that the idiomatic modeling of objects in the Objective Join Calculus has strong connections with the main TSOP features, including state-sensitive operations, explicit state change, and runtime state querying. Such connections draw heavily from the chemical metaphor that inspired the Objective Join Calculus: programs are modeled as chemical soups of molecules (*i.e.* multisets of messages sent to objects) that encode both the current state of the objects and the (pending) operations on them, while reaction rules correspond to objects' methods definitions. In particular, chemical reactions explicitly specify both the valid combinations of state and operations as well as the changes performed by each operation on the state of an object. Incidentally, we observe that the Objective Join Calculus natively supports high-level concepts such as *compound* and *multidimensional* states (Sunshine et al. 2011). This allows us to formally investigate the issues arising when states are partially/concurrently updated.

The second contribution of this paper is a theory of *behavioral types* for TSOP in the Objective Join Calculus and a corresponding *substructural type system*. We exploit the chemical metaphor once more to express in a unified and compositional way the combination of the encapsulated part of objects (state) and their public interface (operations) and to describe objects protocols in terms of the valid configurations of messages that the objects can/must handle. The key idea underpinning the type system is that distinct references to the same object may be given different types. This feature accounts for the fact that several processes may use the same object concurrently. For example, a lock could be shared by two processes P and Q and be acquired by one of them, say P. Then, the reference to the lock held by P would have a type stating that P must (eventually) release the lock, whereas the reference to the lock held by Q would have a type stating that Q can (but need not) attempt to acquire the lock. The *overall* type of the lock would be the

```
def o = FREE | acquire(r) ▷ o.BUSY | r.reply(o)
   or   BUSY | release    ▷ o.FREE
in ( o.FREE
   | def c = reply(o') ▷ o'.release in o.acquire(c) )
```

Listing 1. A lock in the Objective Join Calculus.

combination of these two types, defined in terms of a suitable behavioral connective. The difference between an object's type and the combination of the types of all of its references is explained in terms of a *behavioral subtyping relation*. This relation serves other purposes as well: aside from realizing the obvious form of subtype polymorphism, it is used to characterize safe (and possibly partial) state updates and to derive the protocol of objects with uncertain state. On objects without typestates, subtyping collapses to the traditional one.

With these ingredients in place, we provide a simple static analysis ensuring two key properties of well-typed processes called *respected prohibitions* and *weakly fulfilled obligations*: the former means that well-typed processes comply with the possibilities and prohibitions expressed by the protocols of the objects they use; the latter means that well-typed processes do not discard objects for which they have pending obligations. The enforcement of stronger properties such as deadlock freedom goes beyond the scope of our typing discipline and is left for future work. Compared to the approach of Damiani et al. (2008), our approach enables a fine-grained tuning of the kind of concurrency allowed on objects with structured protocols. For non-aliased objects, we can take full advantage of *static typestate checking* to guarantee that methods are only invoked at the right time. For shared/aliased objects, we can rely on the *runtime synchronization semantics* of the Objective Join Calculus to resolve races and execute methods at the right time. We can thus realize blocking methods *à la* Damiani et al. (2008) or non-blocking methods that inspect and report (some suitable abstraction of) an object's internal state. Overall, as thoroughly exemplified in the rest of the paper, our approach allows us to regulate the balancing between aliased and non-aliased objects and to mix and match static and runtime mechanisms for ensuring usage compliance on concurrent objects with compound states and structured protocols.

Structure of the paper. We start with an informal overview of TSOP in the Objective Join Calculus (Section 2) before recalling its syntax and semantics (Section 3). We present the syntax and semantics of types (Section 4), we describe the rules of the type system (Section 5), and comment on its safety properties (Section 6). In the latter part of the paper, we illustrate a few more advanced examples (Section 7) and the key ingredients needed to implement the proposed framework (Section 8), we discuss related work in more detail (Section 9) and finally hint at future research directions (Section 10). Proofs of the results can be found in Appendix A.

Origin of the material. An early version of this paper appears in the proceedings of OOPSLA 2015 (Crafa and Padovani 2015).

2 THE CHEMISTRY OF TYPESTATES

The Chemical Metaphor. The Join Calculus (Fournet and Gonthier 1996, 2000) originates from the Chemical Abstract Machine (Berry and Boudol 1992), a formal model of computations as sequences of chemical reactions transforming molecules. The Objective Join Calculus (Fournet et al. 2003a) is a mildly sugared version of the Join Calculus with object-oriented features: a program is made of a set of *objects* and a *chemical soup* of messages that can combine into

complex molecules; each object consists of *reaction rules* corresponding to its methods; reaction rules are made of a *pattern* and a *body*: when a molecule in the soup matches the pattern of a reaction, the molecule is consumed and the corresponding body produces other molecules.

Listing 1 shows the idiomatic implementation and use of a *lock* in the Objective Join Calculus. The definition on lines 1–2 creates a new object `o` with two reaction rules, separated by `or`. The symbol ▷ separates the pattern from the body of each rule, while `|` combines messages into complex molecules. The first reaction "fires" if a `FREE` message and an `acquire` message (with argument `r`) are sent to `o`: the two messages are consumed and those on the right hand side of ▷ are produced. In this case, the argument `r` of `acquire` is a reference to another object representing the process that wants to acquire the lock. Hence the effect of triggering the first reaction is that a `BUSY` message is sent to `o` (in jargon, to "self") and a `reply` message is sent to `r` to notify the receiver that the lock has been successfully acquired. The second reaction specifies that the object can also consume a molecule consisting of a `BUSY` message and a `release` message. The reaction just sends a `FREE` message to `o`. The lock is initialized on line 3, by sending a `FREE` message to `o`.

The process on line 4 shows a typical use of the lock. Since communication in the Join Calculus is asynchronous, sequential composition is modeled by means of *continuation passing*: the process creates a continuation object `c` that reacts to the `reply` message sent by the lock; then, the process manifests its intention to acquire the lock by sending `acquire(c)` to `o`. When the reaction on line 1 fires, the `reply` triggers the reaction in `c` on line 4, causing the lock to be released. One aspect not explained in the above description is the passing of `o` in the `reply` message on line 1 which is bound to `o'` on line 4. Since on line 1 `o` corresponds to "self", sending `o` in the message `reply(o)` enables *method chaining*. In fact, with some appropriate syntactic sugar we could rewrite the process on line 4 just as `o.acquire.release`. We will introduce a generalization of such syntactic sugar later on (see Example 3.2 and Listing 3). We will also see that method chaining is not just a trick for writing compact code, but is a key feature that our type system hinges on.

In the next section we will discuss a more complex use case (Example 3.3) where the lock is shared by two processes that compete for acquiring it. In that case, we will see that the complex molecules in the patterns of the lock's reaction rules are essential to make sure that the lock behaves correctly, namely that only one process can hold the lock at any time. In particular, if an `acquire(c')` message is available but there is no `FREE` message in the soup (because another process has previously acquired the lock thereby consuming `FREE`), the reaction in line 1 cannot fire and the process waiting for the `reply` message on `c'` is suspended until the lock is released.

State and Operations in the Join Calculus. Listing 1 provides a clear illustration of TSOP in the Join Calculus: a lock is either free or busy; it can only be acquired when it is free, and it can only be released when it is busy; acquisition makes the lock busy, and release makes it free again. The compound molecules in the patterns specify the valid combinations of state and operations, and the state is explicitly changed within the body of reactions.

These observations lead to a natural classification of messages in two categories: `FREE` and `BUSY` encode the *state* of the lock, while `acquire` and `release` represent its *operations* (we follow the convention that "state" messages are written in upper case and "operation" messages in lower case). Ideally, lock users should not even be aware of the existence of `FREE` and `BUSY`, if only to prevent accidental or malicious violations of the lock protocol. As we will see in Example 5.5, the connectives in our type language allow us to fine-tune the set of messages that can be sent by the users of an object, therefore realizing an implicit encapsulation mechanism for state messages.

Messages in the chemical soup encode the current state of the object and the (pending) operations on it: for instance, the presence of a message `o.FREE` in the soup encodes the fact that the object `o` *is in state* `FREE`; the presence of a message `o.acquire` in the soup encodes the fact that *there is a pending invocation* to the `acquire` method of the object `o`.

Representing state using (molecules of) messages makes it simple to model so-called *and-states* (Harel 1987; Sunshine et al. 2011), of which we will see a few instances at work in Section 7. On the contrary, FREE and BUSY are examples of *or-states* which mutually exclude each other. The typing of the lock object will guarantee that there is always exactly one message among FREE and BUSY, *i.e.* that the state of the lock is always uniquely determined.

Behavioral Types for the Join Calculus. Since in the Join Calculus there is no sharp distinction between (private) messages that encode the object's state and (public) messages that represent the object's operations, we can devise a type language to describe the valid configurations of messages that objects can/must handle. In fact, we can use types to specify (and enforce) the object protocol. Object types are built from message types $\mathtt{m}(\tilde{t})$ using three behavioral connectives, the *product* $\otimes$, the *choice* $\oplus$, and the *exponential* $*$. An object of type $\mathtt{m}(\tilde{t})$ *must* be used by sending an m-tagged message to it with a (possibly empty) tuple of arguments of type $\tilde{t}$; an object of type $t \otimes s$ *must* be used **both** as specified by t **and** as specified by s; an object of type $t \oplus s$ *must* be used **either** as specified by t **or** as specified by s; an object of type $*t$ *can* be used any number of times (even zero), each time as specified by t. To save a few parentheses, we will assume that $*$ binds stronger than $\oplus$ and $\otimes$.

As an example, let us illustrate the type of the lock object. It is useful to keep in mind the intuition that the type of the lock should describe the whole set of valid configurations of messages targeted to the lock. In this respect, we recall that:

- there *must* be one message among FREE and BUSY that represents the state of the lock;
- there *can* be an arbitrary number of acquire messages regardless of the state of the lock (the lock is useful only if it is shared among several processes);
- there *must* be one release message if the lock is BUSY (this is an eventual obligation).

We express all these constraints with the type

$$t_{lock} \stackrel{\text{def}}{=} *\mathtt{acquire(reply(release))} \otimes (\mathtt{FREE} \oplus (\mathtt{BUSY} \otimes \mathtt{release}))$$

It is no coincidence that the only occurrence of $*$ is used in front of the only message (acquire) for which there are no obligations: the lock *can* but need not be acquired. However, if the lock is acquired, then it *must* be released; thus there is no $*$ in front of release. There is no $*$ in front of FREE and BUSY either, meaning that there is an obligation to produce these messages too, but since FREE and BUSY occur in different branches of a $\oplus$ connective, only one of them must be produced. In addition to possibilities and obligations, t_{lock} expresses prohibitions: all message configurations containing multiple FREE or BUSY messages or both FREE and release messages are prohibited by the type. Our type system will guarantee that any lock object is always in a configuration that is legal according to t_{lock}. This implies, for example, that a well-typed process never attempts to release a lock that is in state FREE. In connection with TSOP, the type t_{lock} concisely expresses the fact that the acquire operation is available in both FREE and BUSY states (indeed, it is composed with these state messages using the $\otimes$ connective), whereas release is only available in the BUSY state.

There is one last thing to discuss before we end this informal overview, that is the type of the argument of acquire, named r in Listing 1. If we look at the code, we see that r is the reference to an object to which the lock sends a reply(o) message. Not surprisingly then, the argument of acquire has type reply(release) in t_{lock}. This means that the reference o' in Listing 1 has type release, which is consistent with the way it is used on line 4. In other words, we use method chaining to express the change in the (public) type of an object as methods are invoked. Both o and o' refer to the same lock object, but they have different interfaces: the former can be used for acquiring the lock; the latter must be used (once) for releasing it.

Process	P, Q	::=	$\texttt{null}$	(null process)
		|	$u\,.\,M$	(message sending)
		|	$P \mid Q$	(process composition)
		|	$\texttt{def}\ a = C\ \texttt{in}\ P$	(object definition)
Molecule	M, N	::=	$\mathtt{m}(\tilde{u})$	(message)
		|	$M \mid N$	(molecule composition)
Pattern	J, K	::=	$\mathtt{m}(\tilde{x})$	(message pattern)
		|	$J \mid K$	(pattern composition)
Class	C, D	::=	$J \triangleright P$	(reaction rule)
		|	$C\ \texttt{or}\ D$	(class composition)

Table 1. Syntax of the Objective Join Calculus.

[NULL]	$\Vdash \texttt{null}$	$\rightleftharpoons$	$\Vdash$		
[DEF]	$\mathcal{D} \Vdash \mathcal{P}, \texttt{def}\ a = C\ \texttt{in}\ P$	$\rightleftharpoons$	$\mathcal{D}, a = C \Vdash \mathcal{P}, P$		$a \notin \mathsf{fn}(\mathcal{P})$
[PAR]	$\Vdash P \mid Q$	$\rightleftharpoons$	$\Vdash P, Q$		
[JOIN]	$\Vdash a\,.\,(M \mid N)$	$\rightleftharpoons$	$\Vdash a\,.\,M, a\,.\,N$		
[RED]	$a = \{J_i \triangleright P_i\}_{i\in I} \Vdash a\,.\,\sigma J_k$	$\longrightarrow$	$a = \{J_i \triangleright P_i\}_{i\in I} \Vdash \sigma P_k$		$k \in I$

Table 2. Semantics of the Objective Join Calculus.

3 THE OBJECTIVE JOIN CALCULUS

The syntax of the Objective Join Calculus is defined in Table 1. We assume countable sets of *object names* $a, b, c, \ldots$ and of *variables* $x, y, \ldots$. We let $u, v, \ldots$ denote *names*, which are either object names or variables, and use $\mathtt{m}, \ldots$ to range over *message tags*. We write $\tilde{u}$ for a (possibly empty) tuple $u_1, \ldots, u_n$ of names; we will use this notation extensively for denoting tuples of various entities. Occasionally, we will also use $\tilde{u}$ as the set of names in $\tilde{u}$.

The syntax of the calculus comprises the syntactic categories of *processes*, *molecules*, *patterns*, and *classes*. Molecules are assemblies of messages and each message $\mathtt{m}(\tilde{u})$ is made of a tag $\mathtt{m}$ and a tuple $\tilde{u}$ of arguments; we will abbreviate $\mathtt{m}()$ with $\mathtt{m}$; *join patterns* (or simply patterns) are molecules whose arguments are all variables.

The process $\texttt{null}$ is inert and does nothing. The process $u\,.\,M$ sends the messages in the molecule M to u. The process $P \mid Q$ is the parallel composition of P and Q. Finally, $\texttt{def}\ a = C\ \texttt{in}\ P$ creates a new instance a of the class C. The name a is bound both in C (where it plays the role of "self") and in P. We omit the object initialization clause used by Fournet et al. (2003a) since we are not concerned with privacy aspects. A class is a disjunction of *reaction rules*, which we will often represent as a set $\{J_i \triangleright P_i\}_{i\in I}$. Each rule consists of a *pattern* J_i and a *body* P_i. The variables in J_i are bound in P_i. An instance of P_i is spawned each time a molecule matching J_i is sent to an object that is instance of the class.

We omit the formal definition of free and bound names, which is standard (Fournet et al. 2003a). We write $\mathsf{fn}(P)$ for the set of free names in P and we identify processes up to renaming of bound names. In this paper we use an additional constraint, which is not restrictive and simplifies the type system: we require classes to have no free names other than "self".

We now turn to the operational semantics of the calculus, which describes the evolution of a *solution* $\mathcal{D} \Vdash \mathcal{P}$ made of a set $\mathcal{D} = \{a_i = C_i\}_{i \in I}$ of object definitions and a multiset $\mathcal{P}$ of parallel processes. Intuitively, $\mathcal{P}$ is a "soup" of processes and molecules that is subject to changes in the temperature (expressed by a relation $\rightleftharpoons$) and reactions (expressed by a relation $\rightarrow$). Heating $\rightharpoonup$ breaks things apart, while cooling $\leftharpoondown$ recombines them together, in possibly different configurations. Heating and cooling are reversible transformations of the soup, defined by the first four rules in Table 2: rule [NULL] states that `null` processes may evaporate or condense; rule [DEF] moves objects definitions to/from the $\mathcal{D}$ component of solutions, taking care not to capture free names (disposing of a countable set of object names, we can always silently perform suitable alpha-renamings to avoid captures); rule [PAR] breaks and recombines processes and rule [JOIN] does the same with molecules. To avoid unnecessary clutter, following Fournet et al. (2003a), in all rules except [DEF] we omit unaffected definitions and processes. In rule [DEF], it is important to mention the whole set of definitions $\mathcal{D}$ to make sure that the name a of the object being defined is fresh. Rule [RED] defines reactions as non-reversible transformations of the soup. A reaction may happen whenever the soup contains a molecule targeted to some object a such that the shape of the molecule matches the pattern of one of the rules in the class of a, up to some substitution σ mapping variables to object names (recall that $\{J_i \triangleright P_i\}_{i \in I}$ stands for an `or`-composition of reaction rules, as by Table 1). In this case, the molecule is consumed by the reaction and replaced by the body of the rule, with the substitution σ applied. Note that the heating/cooling rules [PAR] and [JOIN] are key to rearrange molecules so that they can match the pattern of a reaction rule.

Remark 1. The operational semantics presents three forms of non-determinism, due to the heating/cooling of molecules in the soup, the interleaving of reactions pertaining to different objects, and the choice of reactions pertaining to each single object. The first form of non-determinism is only relevant in the formal model. In practice, it is resolved since the Objective Join Calculus enjoys *locality*: each reaction involves messages targeted to the *same* object, therefore all messages sent to an object a travel to and react at the *unique* location of a. The second form of non-determinism accounts for the concurrent setting that we are modeling: when the soup contains molecules that can trigger reactions pertaining different objects, these reactions may fire in any order or even simultaneously, depending on the system architecture. The last form of non-determinism arises when there are enough molecules in the soup to trigger different reactions pertaining the same object. This is usually resolved by the compiler, which translates join patterns into code that processes the messages targeted at one given object according to a deterministic scheduler. In this case, the fact that the formal operational semantics is underspecified (*i.e.* non-deterministic) accounts for all possible implementations of join patterns. ■

In the rest of the section we illustrate the calculus by means of examples. For better clarity, we augment the calculus with conditionals and a few native data types, which can be either encoded or added without difficulties.

Example 3.1 (iterator). Listing 2 shows a possible modeling of an array iterator class in the Objective Join Calculus. Like in object-based languages, the class is modeled as an object `ArrayIterator` providing just one factory method, `new` (line 1), whose arguments are an array `a` and a continuation object `r` to which the fresh instance of the iterator is sent. The iterator itself is an object `o` that can be in one of three states, `INIT`, `SOME`, or `NONE`. States `INIT` and `SOME` have arguments `a` (the array being iterated) and `n` (the index of the current element in the array). `INIT` is a transient state used for initializing the iterator (line 2): the iterator spontaneously moves into either state `SOME` or state `NONE`, depending on whether `n` is smaller than the length `#a` of the array or not. When in state `SOME`, the iterator provides a `next` operation (line 3) for reading the current element `a[n]` of the array and moving onto the next one. Since `n` might

```
1  def ArrayIterator = new(a,r) ▷
2    def o = INIT(a,n)           ▷ if n < #a then o.SOME(a,n) else o.NONE
3       or   SOME(a,n) | next(r) ▷ o.INIT(a,n+1) | r.reply(a[n],o)
4       or   SOME(a,n) | peek(r) ▷ o.SOME(a,n)   | r.some(o)
5       or   NONE      | peek(r) ▷ o.NONE        | r.none(o)
6    in o.INIT(a,0) | r.reply(o)
7  in ...
```

Listing 2. An array iterator.

```
1  def Lock = new(r) ▷
2    def o = FREE | acquire(r) ▷ o.BUSY | r.reply(o)
3       or   BUSY | release    ▷ o.FREE
4    in o.FREE | r.reply(o)
5  in let lock = Lock.new      (* lock : t_ACQUIRE *)
6  in let lock = lock.acquire  (* lock : t_RELEASE *)
7  in lock.release
```

Listing 3. Lock class definition.

be the index of the last element of the array, the iterator transits to state `INIT`, which appropriately re-initializes the iterator. The iterator also provides a `peek` operation that can be used for querying the state of the iterator (lines 4–5). The operation does not change the state of the iterator and sends a message on the continuation `r` with either tag `some` or tag `none`, depending on the internal state of the iterator. ■

Example 3.2 (sequential composition). In this example we see how to encode a sequential composition construct

$$\texttt{let}\ \tilde{y} = u.\texttt{m}(\tilde{v})\ \texttt{in}\ P$$

in the Objective Join Calculus. Intuitively, this construct invokes method `m` on object u with arguments $\tilde{v}$, waits for the results $\tilde{y}$ of the invocation, and continues as P. We let

$$\begin{aligned}\texttt{let}\ \tilde{y} = u.\texttt{m}(\tilde{v})\ \texttt{in}\ P \stackrel{\text{def}}{=} \texttt{def}\ c = \texttt{WAIT}(\tilde{x}) \mid \texttt{reply}(\tilde{y}) \triangleright P\{\tilde{x}/\tilde{w}\}\\ \texttt{in}\ c.\texttt{WAIT}(\tilde{w}) \mid u.\texttt{m}(\tilde{v}, c)\end{aligned}$$

where c and $\tilde{x}$ are fresh, $\tilde{w} = \mathrm{fn}(P) \setminus \tilde{y}$, and $P\{\tilde{x}/\tilde{w}\}$ denotes P where $\tilde{w}$ have been replaced by $\tilde{x}$. The twist in this encoding is that all the free names of P except $\tilde{y}$ are temporarily spilled into a message `WAIT` and then recovered when the callee sends the `reply` message on c. Normally, such spilling is not necessary in the encoding with continuation passing. We do it here to comply with our working assumption that classes have no free names other than "self".

Using this construct we rephrase the code of Listing 1 into that of Listing 3, which also encapsulates the lock definition into the `Lock` class. The re-binding of the `lock` name on lines 5 and 6 is typical of languages with explicit continuations (Gay and Vasconcelos 2010). An actual language would provide either adequate syntactic sugar or a native synchronous method call (Fournet and Gonthier 2000). The types in comments will be described in Section 4. ■

Example 3.3 (dining philosophers). We now discuss an example where the same lock is shared by two concurrent processes. Listing 4 models two philosophers that compete for the same fork when hungry. The fork is created on line 5

```
1  def Philosopher = new(fork) ▷
2    def o = THINK | FORK(f) ▷ o.FORK(f) | let f = f.acquire in o.EAT(f)
3       or   EAT(f)          ▷ o.THINK   | f.release
4    in o.THINK | o.FORK(fork)
5  in let fork = Lock.new  (* fork : t_ACQUIRE *)
6  in Philosopher.new(fork) | Philosopher.new(fork)
```

Listing 4. Two dining philosophers.

and shared by two instances of the `Philosopher` class (line 6). Each philosopher alternates between states `THINK` and `EAT`. In addition, the `FORK` message holds a reference to the shared fork and is meant to be an invariant part of each philosopher's state. Transitions occur non-deterministically: while in state `THINK`, the reaction on line 2 may fire; at that point, the philosopher restores the `FORK` message and attempts to acquire the fork; when the fork is acquired, the philosopher transits into state `EAT`. While in state `EAT`, the philosopher holds a reference `f` to the acquired fork; when the reaction on line 3 fires, the fork is released and the philosopher goes back to state `THINK`. Note that this reaction consumes only part of the philosopher's state, which also comprises the `FORK` message. ■

4 SYNTAX AND SEMANTICS OF TYPES

In this section we define a type language to describe object protocols in terms of the *valid configurations* of messages they accept. *Types* $t, s, \ldots$ are the regular trees (Courcelle 1983) coinductively generated by the productions below:

$$t, s \;::=\; \mathbb{0} \;\mid\; \mathbb{1} \;\mid\; \mathtt{m}(\tilde{t}) \;\mid\; t \oplus s \;\mid\; t \otimes s \;\mid\; *t$$

The *message type* $\mathtt{m}(\tilde{t})$ denotes an object that *must* be used by sending a message to it with tag `m` and arguments of type $\tilde{t}$; when $\tilde{t}$ is the empty tuple, we omit the parentheses altogether. Compound types are built using the behavioral connectives $\oplus$, $\otimes$, and $*$: an object of type $t \oplus s$ *must* be used either according to t or according to s; an object of type $t \otimes s$ *must* be used both according to t and also according to s; an object of type $*t$ *can* be used any number of times, each time according to t. Finally, we introduce the constants $\mathbb{0}$ and $\mathbb{1}$, which respectively represent the empty sum and the empty product. Intuitively, $\mathbb{0}$ is the type of all objects and $\mathbb{1}$ is the type of all objects without obligations. In the examples we will also use basic or array types such as `int`, `real`, or `int[]`.

Here are a few examples: an object of type $\mathtt{m}(\mathtt{int})$ must be used by sending an `m` message to it with one argument of type `int`; an object of type $\mathtt{m}(\mathtt{int}) \oplus \mathbb{1}$ can be used by sending an `m` message to it, or it can be left alone; an object of type $\mathtt{m} \oplus \mathtt{m}'$ must be used by sending either an `m` message or an $\mathtt{m}'$ message to it, while an object of type $\mathtt{m} \otimes \mathtt{m}'$ must be used by sending both an `m` message and an $\mathtt{m}'$ message to it; finally, an object of type $*(\mathtt{m} \oplus \mathtt{m}')$ can be used by sending any number of `m` and $\mathtt{m}'$ messages to it. There is no legal way to use an object of type $\mathbb{0}$.

We do not devise an explicit syntax for recursive types. We work instead with (possibly infinite) regular trees directly and we introduce a family of infinite types as solutions of a finite system of equations. For example, the equation

$$t = \mathbb{1} \oplus \mathtt{m}(t)$$

is satisfied by the infinite (regular) type

$$t \stackrel{\text{def}}{=} \mathbb{1} \oplus \mathtt{m}(\mathbb{1} \oplus \mathtt{m}(\mathbb{1} \oplus \cdots))$$

which denotes an object that can be used by sending an $\mathtt{m}$-tagged message to it with an argument which is itself an object with type t. The shape of the equation, with the metavariable t that occurs unguarded on the left hand side and guarded by (one or more) type constructors on the right hand side, makes sure that t does exist and is uniquely defined. Courcelle (1983) details the metatheory of regular trees, including their relation with finite systems of equations. We require every infinite branch of a type to go through infinitely many message type constructors. This condition (a strengthened contractiveness) excludes types such as $t = t \oplus t$ or $t = *t$ which are meaningless in our setting and provides us with an induction principle on the structure of types that we will use in Definition 4.1 below.

We reserve some notation for useful families of types: we use $\mathtt{M}$ to range over *message types* $\mathtt{m}(\tilde{t})$ and T, S to range over *molecule types*, namely types of the form $\bigotimes_{i\in I} \mathtt{M}_i$; we identify molecule types modulo associativity and commutativity of $\otimes$ and product with $\mathbb{1}$; if $T = \bigotimes_{i\in I} \mathtt{m}_i(\tilde{t}_i)$, we write $\overline{T}$ for its *signature*, namely the multiset $\{\mathtt{m}_i\}_{i\in I}$.

The following definition formalizes the idea that types describe the valid configurations of messages that can be sent to objects. Whenever X and Y are sets of molecule types, we let $\mathsf{X}\cdot\mathsf{Y} \stackrel{\text{def}}{=} \{T \otimes S \mid T \in \mathsf{X} \wedge S \in \mathsf{Y}\}$ and we write X^n for the n-th power of X for $n \in \mathbb{N}$, where $\mathsf{X}^0 = \{\mathbb{1}\}$.

Definition 4.1 (valid configuration). The *interpretation* of a type t, denoted by $[\![t]\!]$, is the set of molecule types inductively defined by the following equations:

$$[\![\mathbb{0}]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![t \oplus s]\!] \stackrel{\text{def}}{=} [\![t]\!] \cup [\![s]\!] \qquad [\![\mathtt{M}]\!] \stackrel{\text{def}}{=} \{\mathtt{M}\}$$
$$[\![\mathbb{1}]\!] \stackrel{\text{def}}{=} \{\mathbb{1}\} \qquad [\![t \otimes s]\!] \stackrel{\text{def}}{=} [\![t]\!] \cdot [\![s]\!] \qquad [\![*t]\!] \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} [\![t]\!]^n$$

We say that T is a *valid configuration* for t if $T \in [\![t]\!]$.

For instance, $[\![\mathtt{m} \oplus \mathtt{m}']\!] = \{\mathtt{m}, \mathtt{m}'\}$ and $[\![\mathtt{m} \otimes \mathtt{m}']\!] = \{\mathtt{m} \otimes \mathtt{m}'\}$. Indeed, the user of an object of type $\mathtt{m} \oplus \mathtt{m}'$ can choose to send *either* $\mathtt{m}$ *or* $\mathtt{m}'$, whereas the user of an object of type $\mathtt{m} \otimes \mathtt{m}'$ must send *both*. Note that $\mathbb{0}$ has no valid configurations, that the only valid configuration of $\mathbb{1}$ is the empty molecule type, and that type $*t$ has, in general, infinitely many valid configurations. For instance, $[\![*\mathtt{m}]\!] = \{\mathbb{1}, \mathtt{m}, \mathtt{m} \otimes \mathtt{m}, \mathtt{m} \otimes \mathtt{m} \otimes \mathtt{m}, \dots\}$.

We have collected all the ingredients for defining the subtyping relation. The intuition behind subtyping is the usual safe substitution principle: when $t \leqslant s$, it is safe to use an object of type t where an object of type s is expected. In our setting, "using an object of type s" means sending to the object a message configuration that is valid for s. For instance, we expect that $\mathtt{m} \oplus \mathtt{m}' \leqslant \mathtt{m}$. The user of an object of type $\mathtt{m}$ must send $\mathtt{m}$ to it. This is also a particular valid use (although not the only valid use) of an object of type $\mathtt{m} \oplus \mathtt{m}'$, which requires its users to send either $\mathtt{m}$ or $\mathtt{m}'$. On the contrary, we expect that $\mathtt{m} \otimes \mathtt{m}' \not\leqslant \mathtt{m}$ and $\mathtt{m} \not\leqslant \mathtt{m} \otimes \mathtt{m}'$. The user of an object of type $\mathtt{m} \otimes \mathtt{m}'$ must send both $\mathtt{m}$ and $\mathtt{m}'$, hence sending only $\mathtt{m}$ is an illegal way of using it. Vice versa, the user of an object of type $\mathtt{m}$ must send only $\mathtt{m}$, hence sending also $\mathtt{m}'$ is an illegal way of using it. We formalize $\leqslant$ resorting to a coinductive definition because types are possibly infinite terms:

Definition 4.2 (subtyping). We write $\leqslant$ for the largest relation between types such that $t \leqslant s$ and $\bigotimes_{i\in I} \mathtt{m}_i(\tilde{s}_i) \in [\![s]\!]$ imply that there is $\bigotimes_{i\in I} \mathtt{m}_i(\tilde{t}_i) \in [\![t]\!]$ such that $\tilde{s}_i \leqslant \tilde{t}_i$ for every $i \in I$. If $t \leqslant s$ holds, then we say that t is a *subtype* of s and s a *supertype* of t. We write $t \simeq s$ if both $t \leqslant s$ and $s \leqslant t$ hold.

According to Definition 4.2, each valid configuration for s must also be a valid configuration for t, up to contravariant subtyping of argument types. More specifically, whenever $S \in [\![s]\!]$, there exists some $T \in [\![t]\!]$ with the same signature as S such that the arguments of corresponding messages in T and S are related contravariantly. For instance, if $s = \mathtt{m}(\mathtt{int})$, then using an object of type s means sending to the object one message of the form $\mathtt{m}(n)$, where n is an integer number. Then, assuming $\mathtt{int} \leqslant \mathtt{real}$, it is safe to replace such object with another one of type $t = \mathtt{m}(\mathtt{real})$: the message $\mathtt{m}(n)$

sent to the former object will be understood without problems also by the latter object, as any integer number is also a real number. Therefore, $\mathtt{m(real)} \leqslant \mathtt{m(int)}$. We also have that $\mathtt{m}(t) \leqslant \mathtt{m}(t \oplus s)$ and $\mathtt{m}(s) \leqslant \mathtt{m}(t \oplus s)$, namely $\mathtt{m}(t \oplus s)$ is an upper bound of both $\mathtt{m}(t)$ and $\mathtt{m}(s)$ (in fact, it is the *least* upper bound of these two types). The user of an object of type $\mathtt{m}(t \oplus s)$ must send $\mathtt{m}$ with an argument that *can* be used according to either t or s, hence this is also a valid use for an object of type $\mathtt{m}(t)$ or an object of type $\mathtt{m}(s)$. We will see a key instance of these relations in Example 4.7.

The interested reader can verify a number of additional properties that will be tacitly used hereafter: that $\mathbb{0}$ and $\mathbb{1}$ are indeed the units of $\oplus$ and $\otimes$; that $\mathbb{0}$ is absorbing for $\otimes$; that $\otimes$ distributes over $\oplus$. We capture all these properties by the following proposition.

Proposition 4.3. *The following properties hold:*

(1) $\leqslant$ *is a pre-order and a pre-congruence;*
(2) *the set of types modulo* $\simeq$ *forms a commutative Kleene algebra (Conway 1971).*

We give a useful taxonomy of types: *linear* types denote objects that *must* be used; *non-linear* types denote objects without obligations; *usable* types denote objects that *can* be used, in the sense that there is a valid way of using them.

Definition 4.4 (type classification). We say that t is *non linear*, written $\mathsf{nl}(t)$, if $t \leqslant \mathbb{1}$; that t is *linear*, written $\mathsf{lin}(t)$, if $t \not\leqslant \mathbb{1}$; that t is *usable*, written $\mathsf{usable}(t)$, if $t \neq \mathbb{0}$.

If $t \leqslant \mathbb{1}$, then $\mathbb{1} \in [\![t]\!]$ namely it is allowed not to send any message to an object of type t. If $t \simeq \mathbb{0}$, then t is linear but not usable, hence it denotes *absurd* objects that must be used, but at the same time such that there is no valid way of using them.

Example 4.5 (standard class type). The class of a conventional object-oriented language containing methods with signatures $\{\mathtt{m}_i(\tilde{t}_i)\}_{i\in I}$ can be described as the type $\bigotimes_{i\in I} *\mathtt{m}_i(\tilde{t}_i)$, saying that the objects of this class can be used for unlimited invocations of all of the available methods, in whatever order. Our subtyping relation is consistent with that typically adopted in such languages, since $\bigotimes_{i\in I} *\mathtt{m}_i(\tilde{t}_i) \leqslant \bigotimes_{j\in J} *\mathtt{m}_j(\tilde{s}_j)$ if and only if $I \supseteq J$ and $\tilde{t}_j \geqslant \tilde{s}_j$ for all $j \in J$ (the subclass has more methods, with arguments of larger type). ■

Example 4.6 (lock interfaces). We illustrate the typing of the lock object used in Listings 1 and 3. Observe that the type t_{lock}, discussed in Section 2, describes the lock object as a whole in terms of both states and operations. Correspondingly, t_{lock} can be correctly assigned to the binding occurrence of `o` on line 2 in Listing 3 according to the type system we will define in Section 5. Lock users are solely concerned with the public interfaces of the lock, which only refer to the `acquire` and `release` methods. We define:

$$t_{\mathsf{ACQUIRE}} \stackrel{\text{def}}{=} *\mathtt{acquire}(\mathtt{reply}(t_{\mathsf{RELEASE}}))$$
$$t_{\mathsf{RELEASE}} \stackrel{\text{def}}{=} \mathtt{release}$$

respectively for the interface of unacquired and acquired locks. Observe that t_{ACQUIRE} is non linear, indicating no obligations on unacquired locks (they can be used any number of times) whereas t_{RELEASE} is linear, indicating that acquired locks must be released (eventually). These interfaces can be "derived" (quite literally) by removing the state types from t_{lock}; we will make the notion of "derivation" precise in Section 5.

The fact that (unacquired) locks can be shared without constraints is a consequence of the relation

$$t_{\mathsf{ACQUIRE}} \simeq t_{\mathsf{ACQUIRE}} \otimes t_{\mathsf{ACQUIRE}}$$

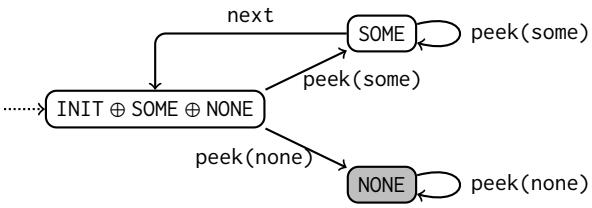

Fig. 1. Transition diagram of the iterator.

stating a well-known property of the exponential/Kleene star. This property is precisely the one needed for typing the code in Listing 4, where one `fork` of type $t_{\texttt{ACQUIRE}}$ is created (line 5) and then shared by two philosophers (line 6). Thanks to this property, the type of a lock is independent of the number of processes trying to acquire it. Note that different references to the same lock, like the references to the fork shared by the two philosophers in Listing 4, may have different types corresponding to the different public interfaces exposed by the references. For instance, the reference `f` held by an eating philosopher has type $t_{\texttt{RELEASE}}$, hence it prescribes a release, while the reference `f` held by a thinking philosopher has type $t_{\texttt{ACQUIRE}}$, allowing acquisitions. ■

Example 4.7 (iterator interfaces). Let us consider the array iterator defined in Listing 2. We postpone the description of the whole type of the iterator object until Section 5, and we discuss here just the public interfaces exposed by the object in the different states, with the help of the transition diagram in Figure 1. When in state `NONE`, the iterator has reached the end of the array and there is only one method available, `peek`, which replies with a `none` message containing the iterator unchanged. Therefore, the public interface of the iterator in state `NONE` is the type satisfying the equation

$$t_{\texttt{NONE}} = \texttt{peek}(\texttt{none}(t_{\texttt{NONE}})) \oplus \mathbb{1}$$

The $\mathbb{1}$ term makes $t_{\texttt{NONE}}$ non linear, allowing the disposal of the iterator when in state `NONE`. Without it, linearity would force us to keep using the iterator even at the end of the iteration. This is depicted in Figure 1 with a shaded box.

The interface of the iterator in state `SOME` must give access to both the `next` and `peek` operations. A tentative type for the iterator in this state is the one satisfying the equation

$$t_{\texttt{SOME}} = \texttt{peek}(\texttt{some}(t_{\texttt{SOME}})) \oplus \texttt{next}(\texttt{reply}(\texttt{int}, t_?))$$

where `peek` replies with a `some` message containing the iterator unchanged, whereas `next` returns the current element of the array being scanned (of type `int`) and the iterator in an updated state. Inspection of Listing 2 reveals that, after a `next` operation, the iterator temporarily moves into state `INIT` and then eventually reaches either state `SOME` or state `NONE`. Therefore, the type $t_?$ exposing the public interface in this unresolved state is obtained as the "intersection" of the interfaces of the two possible states. More precisely, $t_?$ must be a supertype of both $t_{\texttt{NONE}}$ and $t_{\texttt{SOME}}$. It is not difficult to verify that the $\leqslant$-least upper bound of $t_{\texttt{NONE}}$ and $t_{\texttt{SOME}}$ is

$$t_{\texttt{BOTH}} = \texttt{peek}(\texttt{some}(t_{\texttt{SOME}}) \oplus \texttt{none}(t_{\texttt{NONE}}))$$

showing that, when the state of the iterator is uncertain, only `peek` is allowed. Observe also that `peek` has different types depending on whether the state of the iterator is known or not: when the state is known, the type of `peek` is

more precise (only `some` or only `none` is sent); when the state is unknown, the type of `peek` is less precise (either `some` or `none` is sent). Subtyping tunes the precision of the types of objects, according to the knowledge of their state. ■

5 TYPE SYSTEM

We need type environments for tracking the type of the objects used by processes. A *type environment* Γ is a finite mapping from names to types, written $u_1 : t_1, \ldots, u_n : t_n$ or $\tilde{u} : \tilde{t}$ or $\{u_i : t_i\}_{i \in I}$ as convenient. We write $\emptyset$ for the empty environment, $\mathsf{dom}(\Gamma)$ for the domain of Γ, and Γ_1, Γ_2 for the union of Γ_1 and Γ_2, when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$.

Since each object may be used in different parts of a program according to different interfaces, we also need a more flexible environment combination operator than (disjoint) union. The environment in which a process is typed describes how the process uses the objects for which there is a type assignment in the environment. If the *same* object is simultaneously used by two (or more) processes, its type will be the combination (*i.e.*, the product) of all the types it has in the environments used for typing the processes. For example, if some object u is shared by two distinct processes P and Q running in parallel, P uses u according to t and Q uses u according to s, then the parallel composition of P and Q uses u according to $t \otimes s$ overall. If, on the other hand, the object u is used by only one of the two processes, say P, according to t, then it is used according to t also by the parallel composition of P and Q. Formally, we define an operation $\otimes$ for combining type environments, thus:

Definition 5.1 (environment combination). The *combination* of Γ_1 and Γ_2 is the type environment $\Gamma_1 \otimes \Gamma_2$ such that $\mathsf{dom}(\Gamma_1 \otimes \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ defined by:

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

Many substructural type systems define analogous operators for combining type environments. Notable examples are + of Kobayashi et al. (1999) and $\uplus$ of Sangiorgi and Walker (2001).

It is also convenient to extend the subtyping relation to type environments, to ease the application of subsumption. Intuitively, the relation $t \leqslant s$ indicates that an object of type t "has more features" than an object of type s. Similarly, we wish to extend $\leqslant$ to environments so that $\Gamma \leqslant \Delta$ indicates that the environment Γ has more resources with possibly more features than Δ. We must be careful not to introduce in Γ linear resources that are not in Δ, for this would allow processes to ignore objects for which they have obligations. Technically, we allow weakening for non-linear objects only. The extension of $\leqslant$ to type environments is formalized thus:

Definition 5.2 (environment subtyping). We write $\Gamma \leqslant \Delta$ if:

(1) $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma)$, and
(2) $\Gamma(u) \leqslant \Delta(u)$ for every $u \in \mathsf{dom}(\Delta)$, and
(3) $\mathsf{nl}(\Gamma(u))$ for every $u \in \mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Delta)$.

We can then express the fact that an environment Γ only contains non-linear resources by checking whether $\Gamma \leqslant \emptyset$ holds. In this case, we simply write $\mathsf{nl}(\Gamma)$.

With these notions, we can start commenting on the rules of the type system, shown in Table 3. The rules allow deriving various judgments, for processes, molecules, patterns, classes, and solutions.

Typing rules for processes $\boxed{\Gamma \vdash P}$

$$\text{[T-NULL]}\ \frac{}{\emptyset \vdash \texttt{null}} \qquad \text{[T-SEND]}\ \frac{\Gamma \vdash M :: T}{\Gamma \otimes u : T \vdash u\,.\,M} \qquad \text{[T-PAR]}\ \frac{\Gamma_i \vdash P_i \ ^{(i=1,2)}}{\Gamma_1 \otimes \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{[T-OBJECT]}\ \frac{a : t \vdash C \qquad \Gamma, a : t \vdash P}{\Gamma \vdash \texttt{def}\ a = C\ \texttt{in}\ P}$$

$$\text{[T-SUB]}\ \frac{\Delta \vdash P}{\Gamma \vdash P}\ \Gamma \leqslant \Delta$$

Typing rules for molecules $\boxed{\Gamma \vdash M :: T}$

$$\text{[T-MSG-M]}\ \frac{}{\bigotimes_{i=1..n} u_i : t_i \vdash \mathtt{m}(\tilde{u}) :: \mathtt{m}(\tilde{t})}\ \begin{array}{l}\mathsf{usable}(\tilde{t}) \\ \tilde{u} = u_1, \ldots, u_n \\ \tilde{t} = t_1, \ldots, t_n\end{array} \qquad \text{[T-COMP-M]}\ \frac{\Gamma_i \vdash M_i :: T_i \ ^{(i=1,2)}}{\Gamma_1 \otimes \Gamma_2 \vdash M_1 \mid M_2 :: T_1 \otimes T_2}$$

Typing rules for patterns $\boxed{\Gamma \vdash J :: T}$

$$\text{[T-MSG-P]}\ \frac{}{\tilde{x} : \tilde{t} \vdash \mathtt{m}(\tilde{x}) :: \mathtt{m}(\tilde{t})}\ \mathsf{usable}(\tilde{t}) \qquad \text{[T-COMP-P]}\ \frac{\Gamma_i \vdash J_i :: T_i \ ^{(i=1,2)}}{\Gamma_1, \Gamma_2 \vdash J_1 \mid J_2 :: T_1 \otimes T_2}\ \overline{T_1} \cap \overline{T_2} = \emptyset$$

Typing rules for classes $\boxed{u : t \vdash C}$

$$\text{[T-REACTION]}\ \frac{\Gamma \vdash J :: T \qquad \Gamma, a : s \vdash P}{a : t \vdash J \triangleright P}\ \begin{array}{l} t \downarrow T \\ t \leqslant t[T] \otimes s\end{array} \qquad \text{[T-CLASS]}\ \frac{a : t \vdash C_i \ ^{(i=1,2)}}{a : t \vdash C_1\ \texttt{or}\ C_2}$$

Typing rules for solutions $\boxed{\vdash \mathcal{D} \Vdash \mathcal{P}}$

$$\text{[T-DEFINITIONS]}\ \frac{a_i : t_i \vdash C_i \ ^{(i \in I)}}{\{a_i : t_i\}_{i \in I} \vdash \{a_i = C_i\}_{i \in I}} \qquad \text{[T-PROCESSES]}\ \frac{\Gamma_i \vdash P_i \ ^{(i \in I)}}{\bigotimes_{i \in I} \Gamma_i \vdash \{P_i\}_{i \in I}} \qquad \text{[T-SOLUTION]}\ \frac{\Gamma \vdash \mathcal{D} \qquad \Delta \vdash \mathcal{P}}{\vdash \mathcal{D} \Vdash \mathcal{P}}\ \Gamma \leqslant \Delta$$

Table 3. Typing rules.

Rule [T-NULL] states that the idle process is well typed only in an empty environment. Since the idle process does nothing, the absence of linear objects in the environment makes sure that no linear object is left unused. On the other hand, non-linear objects can always be discharged using subsumption [T-SUB], which will be described shortly.

Rule [T-SEND] types message sending $u\,.\,M$, where u is an object and M a molecule of messages. This process is well typed if the type of the object coincides with that of the molecule, which as we will see is just the $\otimes$-composition of the types of the messages in it. Note the use of $\otimes$ in the type environment allowing u to possibly occur in M as the argument of some message.

Rule [T-PAR] types parallel compositions $P_1 \mid P_2$. The rule combines the type environments used for typing P_1 and P_2 to properly keep track of the overall use of the objects shared by the two processes.

Rule [T-OBJECT] types object definitions `def` a = C `in` P. A type t is guessed for the object a and checked to be appropriate for the class C ("appropriateness" will be discussed along with the typing rules for classes) and assigned to a also for typing P. Note that the class C is checked in an environment that contains only a (that is "self"). That is, the type system forces classes to contain no free names other than the reference to self. In principle this is not a restriction, as we have seen in Example 3.2, although in practice it is desirable to allow for more flexibility. We have made this choice to keep the type system as simple as possible. In fact, the type system would remain sound if we allowed C to access non-linear objects. Allowing C to access linear objects is a much more delicate business that requires non-trivial reasoning on the sequence of firings of the rules in C; this is left as a future extension.

Rule [T-SUB] is the subsumption rule, allowing us to enrich the type environment of a process according to Definition 5.2. Intuitively, if P is well typed using the objects described by Δ, then it certainly is well typed in an environment $\Gamma \leqslant \Delta$ where the same objects have more features than those actually used by P and there may be other non-linear objects. This rule is also useful for rewriting the types in the environment as well as for weakening Δ with non-linear objects.

The typing rules for molecules derive judgments of the form $\Gamma \vdash M :: T$. The environment Γ describes the type of the arguments *sent* along the messages in M. The only remarkable feature is the side condition usable($\tilde{t}$) in [T-MSG-M], which requires the arguments of a message to be usable (in the sense of Definition 4.4). This condition is essential for the soundness of the type system (see Example 6.7).

The typing rules for patterns have the form $\Gamma \vdash J :: T$ and are similar to those for molecules. Recall that patterns occur on the left hand side of reaction rules. In this case, the environment Γ describes the type of the arguments *received* when the pattern matches a molecule in the soup. There is a technical difference between [T-COMP-M] and [T-COMP-P]: the former uses the connective $\otimes$ for combining type environments, as it may happen that the same object is sent as argument in different messages; the latter takes the disjoint union of the environments, requiring arguments received from different messages to have different names. The side condition also requires patterns to have disjoint signatures. Overall, variables and message tags occurring in the same pattern must be pairwise distinct. This restriction is typical of most presentations of the Join Calculus and is usually motivated by efficiency reasons: variable linearity avoids the need for equality tests when matching molecules; tag linearity allows the implementation to approximate the state of each message queue with one bit denoting whether the queue is empty or not. In our case, tag linearity is in fact necessary for the soundness of the type system (see Remark 2).

Before looking at the typing rules for classes, let us first consider those for solutions $\mathcal{D} \Vdash \mathcal{P}$, which are essentially unremarkable. Each object definition in $\mathcal{D}$ is typed as in rule [T-OBJECT] and the processes in the multiset $\mathcal{P}$ are typed as if they were all composed in parallel. The two typings are kept consistent by the fact that [T-SOLUTION] uses *related* environments Γ and Δ for both $\mathcal{D}$ and $\mathcal{P}$. The reason why Δ is not exactly Γ is purely technical and accounts for the fact that the subsumption rule [T-SUB] can be applied to the parallel composition of processes $P_1 \mid P_2$, but not after the two processes have been heated and split into the multiset P_1, P_2. More details are provided in Appendix A.

The type system described so far is rather ordinary: the typing rules track the usage of objects and most of the heavy lifting is silently done by subtyping and the $\otimes$ connective. The heart of the type system is [T-REACTION], which verifies that a reaction rule $J \triangleright P$ is appropriate for an object a of type t. The rule determines the type T and bindings Γ of the pattern J and checks that the body P of the rule is well typed in the environment $\Gamma, a : s$. Having a in the environment grants P access to "self". Now, we have to understand which relations should hold among t, T, and s in order for the reaction rule to be safe. In this context "safe" means that:

(1) T describes correctly the type of the received arguments. This is not obvious, because the same tag can be used in messages with arguments of different types while reduction picks messages solely looking at their tag (Table 2). As an example, consider an object of type $t = (\mathtt{A} \otimes \mathtt{m}(\mathtt{int})) \oplus (\mathtt{B} \otimes \mathtt{m}(\mathtt{bool}))$ and observe that the argument of message $\mathtt{m}$ has different types depending on whether the state of the object is $\mathtt{A}$ or $\mathtt{B}$. Then, a reaction $\mathtt{m}(x) \triangleright \cdots$ is unsafe for matching an $\mathtt{m}$-tagged message because it does not provide enough information for understanding whether x has type $\mathtt{int}$ or $\mathtt{bool}$. On the contrary, both $\mathtt{A} \mid \mathtt{m}(x) \triangleright \cdots$ and $\mathtt{B} \mid \mathtt{m}(x) \triangleright \cdots$ are safe reactions, since in these cases the signature of the matched molecule disambiguates the type of x.

(2) By using a according to s, P restores the state of a into one of its valid configurations, described by t. Again this is not obvious, because the only knowledge that P has regarding the state of a comes from the matching of J, which in general is a fraction of all the messages targeted to a at the time of the reaction. As an example, consider an object a of type $t = \mathtt{A} \oplus (\mathtt{B} \otimes \mathtt{m})$. Then, a reaction $\mathtt{A} \triangleright a.\mathtt{B} \mid a.\mathtt{m}$ is safe and so is a reaction $\mathtt{B} \mid \mathtt{m} \triangleright a.\mathtt{A}$. In both cases, the reaction consumes and produces a valid configuration of t. On the contrary, a reaction $\mathtt{B} \triangleright a.\mathtt{A}$ is unsafe. Indeed, from t we know that when a message $\mathtt{B}$ is present in a valid configuration for t, namely when the reaction can fire, there is (or there will be, eventually) also a message $\mathtt{m}$. Thus, by consuming $\mathtt{B}$ and producing $\mathtt{A}$, the reaction moves the object into a configuration $\mathtt{A} \otimes \mathtt{m}$ which is invalid according to t.

Condition (1) is verified by the side condition $t \downarrow T$ of [T-REACTION], saying when a given molecule type T is not ambiguous in t:

Definition 5.3 (unambiguous pattern). We say that T is *unambiguous* in t, notation $t \downarrow T$, if $\{S \mid S \otimes R \in [\![t]\!] \wedge \overline{S} = \overline{T}\} = \{T\}$.

In words, $t \downarrow T$ holds if for each valid configuration $S \otimes R$ of t that includes a molecule type S sharing the same signature as T, the molecule type is exactly T. In addition, there must be a valid configuration of t that includes T. This implies that t is usable whenever $t \downarrow T$ holds for some T. Recalling the examples made when describing condition (1), if $t = (\mathtt{A} \otimes \mathtt{m}(\mathtt{int})) \oplus (\mathtt{B} \otimes \mathtt{m}(\mathtt{bool}))$, then we have $t \downarrow \mathtt{A} \otimes \mathtt{m}(\mathtt{int})$ and $t \downarrow \mathtt{A} \otimes \mathtt{m}(\mathtt{bool})$, but not $t \downarrow \mathtt{m}(\mathtt{bool})$.

Remark 2. Let be $t = (\mathtt{m}(\mathtt{foo}) \otimes \mathtt{m}(\mathtt{bar})) \oplus \mathbb{1}$ and $T = \mathtt{m}(\mathtt{foo}) \otimes \mathtt{m}(\mathtt{bar})$, then $t \downarrow T$ holds because t has only one valid configuration with the same signature as T. Now consider the reaction $J \triangleright x.\mathtt{foo} \mid y.\mathtt{bar}$ where $J = \mathtt{m}(x) \mid \mathtt{m}(y)$; if we let the judgement

$$x : \mathtt{foo}, y : \mathtt{bar} \vdash J :: T$$

to be derivable even though $\mathtt{m}$ occurs twice in J, there would be no guarantee that, once J matches a molecule, x is actually bound to the argument of type $\mathtt{foo}$ and y is actually bound to the argument of type $\mathtt{bar}$, and not vice versa. For this reason, tag linearity in patterns is a key restriction in our type system, where messages with the same tag can have arguments with different types. ■

Condition (2) is verified by the side condition $t \leqslant t[T] \otimes s$ of [T-REACTION], where the type $t[T]$ represents the "residual" of t after a molecule with type T (the pattern of the reaction rule) has been removed; such residual is combined (in the sense of $\otimes$) with s, which is what P sends to the object; the resulting type $t[T] \otimes s$ is compatible with the object's type t if it is a supertype of t. The type residual operator is defined thus:

Definition 5.4 (type residual). The *residual* of t with respect to $\mathtt{M}$, written $t[\mathtt{M}]$, is inductively defined as follows:

$$\begin{array}{lll} \mathbb{0}[\mathtt{M}] = \mathbb{1}[\mathtt{M}] = \mathbb{0} & & (t \oplus s)[\mathtt{M}] = t[\mathtt{M}] \oplus s[\mathtt{M}] \\ \mathtt{m}(\tilde{t})[\mathtt{m}'(\tilde{s})] = \mathbb{0} & \text{if } \mathtt{m} \neq \mathtt{m}' & (t \otimes s)[\mathtt{M}] = (t[\mathtt{M}] \otimes s) \oplus (t \otimes s[\mathtt{M}]) \\ \mathtt{m}(\tilde{t})[\mathtt{m}(\tilde{s})] = \mathbb{1} & & (*t)[\mathtt{M}] = t[\mathtt{M}] \otimes *t \end{array}$$

We extend the residual to molecule types in the obvious way, that is $t[\mathbb{1}] = t$ and $t[\mathtt{M} \otimes T] = t[\mathtt{M}][T]$.

Note that the type residual operator (Definition 5.4) is nothing but the *Brzozowski derivative* (Brzozowski 1964; Conway 1971) adapted to a commutative Kleene algebra over message types.

To further illustrate the side condition, we work out a few more examples in which we consider different objects a of type t and we write $T \triangleright s$ for denoting a reaction $J \triangleright P$ where J has type T and P is typed in an environment that includes $a : s$. We will say that $T \triangleright s$ is valid or invalid depending on whether the condition holds or not.

- If $t \stackrel{\text{def}}{=} (\mathtt{A} \otimes \mathtt{m}) \oplus \big(\mathtt{B} \otimes (\mathbb{1} \oplus \mathtt{m})\big)$, then $\mathtt{A} \triangleright \mathtt{B}$ is valid but $\mathtt{B} \triangleright \mathtt{A}$ is not. We have $t[\mathtt{B}] = \mathbb{1} \oplus \mathtt{m}$ and $t \not\leqslant (\mathbb{1} \oplus \mathtt{m}) \otimes \mathtt{A}$. In general, the transition from a state in which a message is linear ($\mathtt{m}$) to another where the message is not linear ($\mathbb{1} \oplus \mathtt{m}$) cannot be reversed, because the object may have been discarded or aliased.
- If $t \stackrel{\text{def}}{=} \mathtt{A} \oplus (\mathtt{B} \otimes *\mathtt{foo}) \oplus (\mathtt{C} \otimes *\mathtt{foo} \otimes *\mathtt{bar})$, then $\mathtt{A} \triangleright \mathtt{B}$ and $\mathtt{B} \triangleright \mathtt{C}$ are valid, but neither $\mathtt{B} \triangleright \mathtt{A}$ nor $\mathtt{C} \triangleright \mathtt{B}$ is. It is unsafe for the object to move from state $\mathtt{C}$ to state $\mathtt{B}$ because there could be residual $\mathtt{bar}$ messages not allowed in state $\mathtt{B}$. In general, non-linear messages such as $\mathtt{foo}$ and $\mathtt{bar}$ can only accumulate monotonically across state transitions.
- If $t \stackrel{\text{def}}{=} (\mathtt{A} \otimes \mathtt{m}(\mathtt{int})) \oplus (\mathtt{B} \otimes \mathtt{m}(\mathtt{real}))$, then $\mathtt{A} \triangleright \mathtt{B}$ is valid, but $\mathtt{B} \triangleright \mathtt{A}$ is not. Indeed $t[\mathtt{B}] = \mathtt{m}(\mathtt{real})$ and $t \not\leqslant \mathtt{m}(\mathtt{real}) \otimes \mathtt{A}$. The transition $\mathtt{A} \triangleright \mathtt{B}$ is safe because the $\mathtt{int}$ argument of message $\mathtt{m}$ in state $\mathtt{A}$ can be subsumed to $\mathtt{real}$ in state $\mathtt{B}$, but not vice versa.

Example 5.5 (lock). We illustrate the type system at work showing that the two reactions of the lock (lines 2–3 in Listing 3) and its initialization (line 4) are well typed using the types we discussed in Section 2 and Example 4.6, that is: $t_{lock} = t_{\mathtt{ACQUIRE}} \otimes \big(\mathtt{FREE} \oplus (\mathtt{BUSY} \otimes t_{\mathtt{RELEASE}})\big)$ where $t_{\mathtt{ACQUIRE}} = *\mathtt{acquire}(\mathtt{reply}(t_{\mathtt{RELEASE}}))$ and $t_{\mathtt{RELEASE}} = \mathtt{release}$. Consider the first reaction; for its pattern we derive

$$\mathtt{r} : \mathtt{reply}(t_{\mathtt{RELEASE}}) \vdash \mathtt{FREE} \mid \mathtt{acquire}(\mathtt{r}) :: T$$

where $T \stackrel{\text{def}}{=} \mathtt{FREE} \otimes \mathtt{acquire}(\mathtt{reply}(t_{\mathtt{RELEASE}}))$. Let $s \stackrel{\text{def}}{=} \mathtt{BUSY} \otimes t_{\mathtt{RELEASE}}$, then for the body of the reaction we derive

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \mathtt{BUSY} :: \mathtt{BUSY}}\,[\text{T-MSG-M}]}{\mathtt{o} : \mathtt{BUSY} \vdash \mathtt{o.BUSY}}\,[\text{T-SEND}] \qquad \dfrac{\dfrac{}{\mathtt{o} : t_{\mathtt{RELEASE}} \vdash \mathtt{reply(o)} :: \mathtt{reply}(t_{\mathtt{RELEASE}})}\,[\text{T-MSG-M}]}{\mathtt{r} : \mathtt{reply}(t_{\mathtt{RELEASE}}), \mathtt{o} : t_{\mathtt{RELEASE}} \vdash \mathtt{r.reply(o)}}\,[\text{T-SEND}]}{\mathtt{r} : \mathtt{reply}(t_{\mathtt{RELEASE}}), \mathtt{o} : s \vdash \mathtt{o.BUSY} \mid \mathtt{r.reply(o)}}\,[\text{T-PAR}]$$

Now $t_{lock} \downarrow T$ holds and furthermore

$$t_{lock} \leqslant t_{lock}[T] \otimes s = t_{\mathtt{ACQUIRE}} \otimes \mathtt{BUSY} \otimes t_{\mathtt{RELEASE}}$$

hence the side conditions of [T-REACTION] are satisfied. For the pattern in the second reaction we derive

$$\vdash \mathtt{BUSY} \mid \mathtt{release} :: \mathtt{BUSY} \otimes \mathtt{release}$$

and it is easy to see that the body of the reaction is also well typed. Now, we have $t_{lock} \downarrow \mathtt{BUSY} \otimes \mathtt{release}$ and

$$t_{lock} \leqslant t_{lock}[\mathtt{BUSY} \otimes \mathtt{release}] \otimes \mathtt{FREE} \simeq t_{\mathtt{ACQUIRE}} \otimes \mathtt{FREE}$$

so the side conditions of [T-REACTION] are again satisfied, this time taking $s \stackrel{\text{def}}{=} \mathsf{FREE}$.

Concerning the lock initialization (line 4), we obtain

$$\mathtt{r} : \mathtt{reply}(t_{\mathsf{ACQUIRE}}), \mathtt{o} : t_{lock} \vdash \mathtt{o.FREE} \mid \mathtt{r.reply(o)}$$

with a derivation analogous to that for the body of the first reaction and using the subtyping relation $t_{lock} \leqslant t_{\mathsf{ACQUIRE}} \otimes \mathsf{FREE}$. Overall, we observe that FREE and BUSY are always produced either during the initialization or by the lock itself whereas the public interfaces of the lock (t_{ACQUIRE} and t_{RELEASE}) never allow the user to send these messages directly. In this sense, FREE and BUSY are encapsulated within the lock. ■

Example 5.6 (iterator). We conclude the typing of the array iterator in Example 3.1 (Listing 2). By composing the public interfaces $t_{\mathsf{NONE}}, t_{\mathsf{SOME}}, t_{\mathsf{BOTH}}$ defined in Example 4.7, we can define the type of the iterator object o as follows:

$$t_{iter} \stackrel{\text{def}}{=} (\mathtt{INIT}(\mathtt{int[]}, \mathtt{int}) \otimes t_{\mathsf{BOTH}}) \oplus (\mathtt{SOME}(\mathtt{int[]}, \mathtt{int}) \otimes t_{\mathsf{SOME}}) \oplus (\mathsf{NONE} \otimes t_{\mathsf{NONE}})$$

Notice that t_{iter} is obtained as a disjunction of three types, each corresponding to a pair encoding a possible state and the public interface of the iterator in that state.

In order to check the typing of the definition of the object o in Listing 2, we have to check four reactions; we just discuss two of them and, for readability, we only consider message tags omitting argument types. The first reaction $\mathtt{INIT} \triangleright \mathsf{SOME} \oplus \mathsf{NONE}$ is valid since $t_{iter}[\mathtt{INIT}] = t_{\mathsf{BOTH}}$ and

$$t_{iter} \leqslant (\mathsf{SOME} \oplus \mathsf{NONE}) \otimes t_{\mathsf{BOTH}} \simeq (\mathsf{SOME} \otimes t_{\mathsf{BOTH}}) \oplus (\mathsf{NONE} \otimes t_{\mathsf{BOTH}})$$

because $t_{\mathsf{SOME}} \leqslant t_{\mathsf{BOTH}}$ and $t_{\mathsf{NONE}} \leqslant t_{\mathsf{BOTH}}$ as we have argued in Example 4.7. The reaction $\mathsf{SOME} \otimes \mathtt{next} \triangleright \mathtt{INIT} \otimes t_{\mathsf{BOTH}}$ is also valid since $t_{iter}[\mathsf{SOME} \otimes \mathtt{next}] = \mathbb{1}$ and now

$$t_{iter} \leqslant \mathbb{1} \otimes \mathtt{INIT} \otimes t_{\mathsf{BOTH}} \simeq \mathtt{INIT} \otimes t_{\mathsf{BOTH}}$$

Observe that the code in Listing 2 does not contain any reaction involving both the state INIT *and* the operation peek, since the iterator in state INIT eventually moves into either state SOME or NONE; nevertheless t_{iter} exposes the interface t_{BOTH} while in state INIT, instead of the empty interface. This is because, in lines 6 and 9, a reference o to the iterator is returned to the caller while the iterator is moving to state INIT. Such reference could be used by a quick caller to send a peek message to the iterator while the iterator is still in the transient state INIT, and this requires $\mathtt{INIT} \otimes \mathtt{peek}$ to be a valid configuration of t_{iter}. It is possible to make sure that the reference o returns to the caller only once the iterator has moved away from state INIT, by reshaping INIT into a synchronous operation. ■

6 PROPERTIES OF WELL-TYPED PROCESSES

In this section we prove the key properties enjoyed by well-typed processes. To begin with, we state a completely standard, yet fundamental result showing that typing is preserved under heating, cooling, and reductions.

THEOREM 6.1 (SUBJECT REDUCTION). *If* $\vdash \mathcal{D} \Vdash \mathcal{P}$ *and*

$$\mathcal{D} \Vdash \mathcal{P} \quad \mathcal{R} \quad \mathcal{D}' \Vdash \mathcal{P}'$$

where $\mathcal{R} \in \{\rightharpoonup, \rightharpoondown, \rightarrow\}$, *then* $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.

Theorem 6.1 is crucial for the next results, since it assures that the properties enjoyed by well-typed processes are invariant under arbitrarily long process reductions.

The first proper soundness result states that a well-typed process respects the prohibitions expressed by the types of the objects it manipulates. We say that t *prohibits* T if not usable$(t[T])$, namely if there is no valid configuration of t that includes (a molecule with the same signature as) T. Now we have:

Theorem 6.2 (respected prohibitions). *If*

$$\Gamma, a : t \vdash P \mid a.\mathtt{m}_1(\tilde{c}_1) \mid \cdots \mid a.\mathtt{m}_n(\tilde{c}_n)\,,$$

then usable$(t[\mathtt{m}_1 \otimes \cdots \otimes \mathtt{m}_n])$.

In words, if the type of an object prohibits invocation of a particular method when the object is in some particular state, then there is no well-typed soup of processes containing pending invocations to that method when the object is in that state. To illustrate, consider the lock object in Listing 1. The type t_{lock} of the lock we have defined in Section 2 prohibits invocation of method `release` when the lock is in state `FREE`, indeed $t_{lock}[\mathtt{FREE} \otimes \mathtt{release}] \simeq \mathbb{0}$, that is $\neg$usable$(t_{lock}[\mathtt{FREE} \otimes \mathtt{release}])$. Now, a free lock is identified by the presence of a `o.FREE` molecule in the solution. Hence, the following judgment is *not* derivable

$$\Gamma, \mathtt{o} : t_{lock} \vdash P \mid \mathtt{o.FREE} \mid \mathtt{o.release}$$

Similarly, the type t_{lock} states that, when in state `BUSY`, there can be exactly one pending invocation to `release`. In particular, $t_{lock}[\mathtt{BUSY} \otimes \mathtt{release} \otimes \mathtt{release}] \simeq \mathbb{0}$. So,

$$\Gamma, \mathtt{o} : t_{lock} \vdash P \mid \mathtt{o.BUSY} \mid \mathtt{o.release} \mid \mathtt{o.release}$$

is another judgment that cannot be derived. Remarkably, we can infer a great deal of information regarding the state of an object by solely looking at its type, knowing virtually nothing about the rest of the (well-typed) program. For instance, no soup containing both a `FREE` and a `BUSY` message simultaneously targeted to the same lock is well typed, meaning that the state of every lock is always uniquely determined.

The second soundness result states that a well-typed process fulfills all the obligations with respect to the objects it owns. More precisely, if a process P is typed in an environment that contains a linear object a, that is an object whose type mandates the (eventual) invocation of a particular method, then a cannot be discarded by P, but must be held by P and used according to its type.

Theorem 6.3 (weakly fulfilled obligations). *If* $\Gamma \vdash P$ *and* $a \in \mathsf{dom}(\Gamma)$ *and* $\mathsf{lin}(\Gamma(a))$*, then* $a \in \mathsf{fn}(P)$.

Another way of reading this theorem is that well-typed processes can only discard non-linear objects, namely objects for which they have no pending obligations. For example, since t_{lock} mandates the invocation of method `release` once the lock has been acquired, omitting the `f.release` from line 5 in Listing 4 would result into an ill-typed philosopher.

We have labeled Theorem 6.3 "weak" obligation fulfillment because the property may indeed look weaker than desirable. One would probably expect a stronger property saying that every method that must be invoked *is* eventually invoked. Such stronger property, which is in fact a *liveness* property, is however quite subtle to characterize and hard to enforce with a type system. In particular, it would require well-typed processes to be free from both deadlocks and livelocks, which is something well beyond the capabilities of the type system we have presented in Section 5. The next two examples illustrate why this is the case.

Example 6.4 (deadlock). Assuming a `Lock` class defined as in Listing 3, the following code attempts at acquiring the *same* lock twice, resulting in a deadlock:

```
1  def Lock = ...                  (* see Listing 3 *)
2  in let lock  = Lock.new         (* lock  : t_ACQUIRE *)
3  in let lock1 = lock.acquire     (* lock1 : t_RELEASE *)
4  in let lock2 = lock.acquire     (* lock2 : t_RELEASE *)
5  in lock1.release | lock2.release
```

A `lock` is created on line 2 and used twice on lines 3 and 4 for acquisition. This is possible because of the relation $t_{\mathsf{ACQUIRE}} \simeq t_{\mathsf{ACQUIRE}} \otimes t_{\mathsf{ACQUIRE}}$. Clearly, only the first acquisition succeeds, and the program blocks while performing the second one. Note that the program is well typed, as both `lock1` and `lock2`, which have a linear type, are used in line 5 for releasing the lock, according to the lock protocol. However, such "usage" is merely syntactic, for neither of the two `release` messages will ever be received, and the lock will never be released. Note that static deadlock detection is undecidable in general and non-trivial to approximate. In this example, for instance, it would require understanding that the `acquire` method is a *blocking* one (this information cannot be inferred merely from the type of `Lock`) and that it is the *same* lock being acquired twice, in a fragment of sequential code (this is easy to detect here since `lock` syntactically occurs twice, but in general the code could invoke the `acquire` method on distinct variables that are eventually instantiated with the same reference to `lock`). ■

Example 6.5 (livelock). There is one trivial way to honor all pending obligations (as by Theorem 6.3), namely postponing them forever. For example, let

$$\mathit{forever}(u) \stackrel{\text{def}}{=} \texttt{def}\ c = \mathtt{m}(x) \triangleright c.\mathtt{m}(x)\ \texttt{in}\ c.\mathtt{m}(u)$$

where c is a fresh name. The judgment $a : t \vdash \mathit{forever}(a)$ is derivable for any t such that $\mathsf{usable}(t)$. In particular, t may be linear, and yet $\mathit{forever}(a)$ never invokes any method on a. Although $\mathit{forever}(a)$ fools the type system into believing that all pending obligations on a have been honored, processes like $\mathit{forever}(a)$ are sufficiently contrived to be rarely found in actual code. In other words, we claim that Theorem 6.3 provides practically useful guarantees about the actual use of objects with linear types. ■

Finally, we draw the attention on a general property of the type system that is key for proving Theorem 6.2:

Lemma 6.6. *There exist no* Γ *and* P *such that* $\Gamma, u : \mathbb{0} \vdash P$.

This property states that there is no well-typed process that can hold an unusable object. The result may look obvious, but it has important consequences: we have remarked the role of subtyping for deducing the interface of objects with uncertain state. For instance, t_{BOTH} (Example 4.7) is obtained as the *least upper bound* of t_{NONE} and t_{SOME}. Since $\mathbb{0}$ is the top type, the least upper bound of two (or more) types *always* exists, but it can be $\mathbb{0}$. For example, had we forgotten to equip the iterator with a `peek` operation in state `SOME` (line 4 of Listing 2), t_{BOTH} would be $\mathbb{0}$ and the iterator would be essentially unusable. Lemma 6.6 tells us that the type system detects such mistakes.

Example 6.7. The side condition in rule [T-MSG-M] requires the arguments of a message to have a usable type. If this condition were not enforced it would be possible to derive $a : \mathbb{0} \vdash \mathit{forever}(a)$ in spite of the fact that a has an unusable type (note that $\mathit{forever}(a)$ sends a as the argument of a message). Then, the following derivation would be legal and

Theorem 6.2 would not hold:

$$\dfrac{\dfrac{\dfrac{\vdots}{a : \mathbb{0} \vdash \mathit{forever}(a)} \qquad \dfrac{\dfrac{}{\vdash \mathsf{bar} :: \mathsf{bar}}\,[\text{T-MSG-M}]}{a : \mathsf{bar} \vdash a.\mathsf{bar}}\,[\text{T-SEND}]}{a : \mathbb{0} \otimes \mathsf{bar} \vdash \mathit{forever}(a) \mid a.\mathsf{bar}}\,[\text{T-PAR}]}{a : \mathsf{foo} \vdash \mathit{forever}(a) \mid a.\mathsf{bar}}\,[\text{T-SUB}]$$

Since $\mathsf{foo} \leqslant \mathbb{0} \otimes \mathsf{bar} \simeq \mathbb{0}$, the subsumption rule could be used for allowing prohibited method invocations (bar) knowing that these would be absorbed by $\mathbb{0}$ types in other parts of the derivation. ■

7 EXAMPLES

In this section we discuss a few more advanced examples to illustrate the expressiveness of our approach, which encompasses both statically enforced guarantees and dynamic support from the runtime system. We conclude the section with a summary of the three programming patterns used throughout the paper for realizing state-changing methods in our approach.

7.1 One-Place Buffer

It is possible to assign different types to a given object, corresponding to different usage protocols involving possibly different numbers of intended users. Such degree of polymorphism is made possible by the semantics of the Objective Join Calculus, which allows sending arbitrary configurations of messages to objects and relies on runtime join pattern matching for triggering reactions.

The code fragment below models a one-place buffer as an object with two operations `insert` and `remove` that switch the buffer's state from `EMPTY` to `FULL` and vice versa.

```
def Buffer = new(r) ▷
  def o = EMPTY   | insert(x,r) ▷ o.FULL(x) | r.reply(o)
  or      FULL(x) | remove(r)   ▷ o.EMPTY   | r.reply(x,o)
  in o.EMPTY | r.reply(o)
in ...
```

When used in a single-threaded way, the one-place buffer has just one user at any time, which must necessarily alternate `insert` and `remove` operations in order to achieve a sensible behavior. In this case, the public interface of the buffer is defined by the types t_{EMPTY} and t_{FULL} that satisfy the equations

$$t_{\mathsf{EMPTY}} = \mathtt{insert}(\mathtt{int}, \mathtt{reply}(t_{\mathsf{FULL}})) \oplus \mathbb{1} \qquad\qquad t_{\mathsf{FULL}} = \mathtt{remove}(\mathtt{reply}(\mathtt{int}, t_{\mathsf{EMPTY}}))$$

and the overall type of a buffer is

$$t_{\mathit{buffer}} \stackrel{\text{def}}{=} (\mathsf{EMPTY} \otimes t_{\mathsf{EMPTY}}) \oplus (\mathsf{FULL}(\mathtt{int}) \otimes t_{\mathsf{FULL}})$$

The use of $\mathbb{1}$ just in t_{EMPTY} and not also in t_{FULL} means that it is *possible* to insert an element in an empty buffer and that it is *mandatory* to remove the element from a full buffer. Different combinations of possibilities and obligations can be achieved by a suitable placement of $\mathbb{1}$. A more radical choice concerns the number of threads allowed to access the buffer: the buffer could be simultaneously accessed by both a *producer* thread (which inserts elements in the buffer) and a *consumer* thread (which removes them). In this case, producer and consumer would use the buffer according to the

interfaces defined by

$$t_{prod} = \texttt{insert}(\texttt{int}, \texttt{reply}(t_{prod})) \qquad t_{cons} = \texttt{remove}(\texttt{reply}(\texttt{int}, t_{cons}))$$

When two independent threads access the same buffer, none of them can know with absolute certainty in which state the buffer is. However, according to the chemical semantics of the Join Calculus, if an operation is invoked when the buffer is in a state that disallows the triggering of the corresponding reaction, the invocation remains pending until the buffer moves into the "right" state. For example, unlike the single-thread case, the consumer might issue a `remove` message when the buffer is `EMPTY`, in which case it would not receive an answer until the producer issues an `insert`. In this scenario, the type of the buffer must be revised to account for the possibility that an `insert` message is sent when the buffer is `FULL`, or a `remove` message is sent when the buffer is `EMPTY`:

$$t'_{buffer} \stackrel{\text{def}}{=} (\texttt{EMPTY} \oplus \texttt{FULL}(\texttt{int})) \otimes t_{prod} \otimes t_{cons}$$

Since the buffer state message is *consumed* when a reaction is triggered, producer and consumer always operate in mutual exclusion even if they attempt to access the buffer simultaneously. In Sections 7.2 and 7.3 we will see examples of concurrent objects where several processes may concurrently access the same object when they act on disjoint components of the object's state.

An even more permissive usage policy for the buffer is to allow an arbitrary number of producers and/or consumers. For example, the types

$$(\texttt{EMPTY} \oplus \texttt{FULL}(\texttt{int})) \otimes *t_{prod} \otimes t_{cons} \qquad \text{and} \qquad (\texttt{EMPTY} \oplus \texttt{FULL}(\texttt{int})) \otimes t_{prod} \otimes *t_{cons}$$

respectively account for arbitrarily many producers and one consumer, or for one producer and arbitrarily many consumers. Note that all these different typings are legal for the same object definition (*cf.* rule [T-CLASS] and [T-REACTION]), ultimately realizing a form of code polymorphism. Choosing one type over another means trading flexibility and efficiency: on the one hand, t_{buffer} offers stronger guarantees on the usage protocol of the buffer and allows for a more efficient code (each operation is allowed precisely when the buffer is in the right state to execute it), but limits sharing and concurrent access to the buffer; on the other hand, a type like t'_{buffer} allows more liberal and concurrent usage of the buffer but crucially relies on the runtime support for suspending the operations until the buffer is in a state that allows them to execute.

7.2 Concurrent Queue

In this section we model a non-blocking concurrent queue, showing that by breaking down the state of an object into smaller components we can enhance concurrency of the program. Unlike the one-place buffer of Section 7.1, where concurrent producer and consumer may compete for the buffer but they eventually access it in mutual exclusion, the concurrent queue we model here allows for simultaneous access and modification, whereby the consumer may dequeue an element of the queue while the producer is enqueuing another one. We implement the non-blocking queue following Michael and Scott's concurrent queue algorithms (Michael and Scott 1996): the linked list storing the queue's elements always contains at least one node that stores no sensible data and whose only purpose is to *separate* producer and consumer so that they never interfere except, possibly, when the queue is empty.

The overall state of the queue consists of two references `h` and `t` to the nodes at its head and tail, respectively. In principle, we could store both `h` and `t` in a single state message, say `LIST(h,t)`, but doing so would prevent concurrent access to the queue, since concurrent invocations of `enqueue` and `dequeue` would compete for consuming the sole `LIST`

```
1  def Queue = new(r) ▹
2    def o = TAIL(t) | enqueue(x,r) ▹
3              let node = Node.new(x) in
4              t.link(node) | o.TAIL(node) | r.reply(o)
5         or HEAD(h) | dequeue(r)   ▹
6              case h.has_next() of
7                 no(h)  ▹ o.HEAD(h) | r.none(o)
8              or yes(h) ▹ let k = h.unlink() in
9                          let x,k = k.get_data() in
10                         o.HEAD(k) | r.some(x,o)
11   in let node = Node.new(-1) in
12      o.HEAD(node) | o.TAIL(node) | r.reply(o)
13
14 def Node = new(x,r) ▹
15   def o = DATA(v) | get_data(r) ▹ o.DATA(v) | r.reply(v,o)
16        or LAST    | has_next(r) ▹ o.LAST | r.no(o)
17        or LAST    | link(n)     ▹ o.NEXT(n)
18        or NEXT(n) | has_next(r) ▹ o.NEXT(n) | r.yes(o)
19        or NEXT(n) | unlink(r)   ▹ o.LAST | r.some(n)
20   in o.DATA(x) | o.LAST | r.reply(o)
```

Listing 5. Modeling of a non-blocking, concurrent queue.

message to fire the corresponding reaction. Given that enqueue only modifies the tail of the queue, while dequeue possibly modifies only the head of the queue, it makes sense to *split* the overall state of the queue using two distinct messages HEAD and TAIL, each carrying a reference to the corresponding end of the linked list. In general, splitting compound states into independent messages may enhance concurrency by reducing the synchronizations between independent operations of an object.

The code of the concurrent queue is shown in the upper part of Listing 5 and heavily relies on the syntactic sugar introduced in Example 3.2. The *separator* node (created on line 11) initially contains an arbitrary element -1 and is used for initializing the state of the queue (line 12). When the producer enqueues some data x (reaction on line 2), a new node for x is created (line 3) and the node t at the tail of the queue is linked to node, which becomes the new tail (line 4). When the consumer attempts to dequeue some data (reaction on line 5), the separator h at the head of the queue is queried to check whether it is linked to a subsequent node (line 6). The case u.m($\tilde{v}$) of $\{J_i \triangleright P_i\}_{i \in I}$ construct is a straightforward generalization of let (Example 3.2) for which we omit a formal definition. If the separator is not linked to another node, then the queue is empty and the consumer is notified with a none message (line 7). If the separator is indeed linked to a subsequent node, then *that* node (and not h) is the first proper node of the queue. In this case, the separator h is detached from its successor k (line 8), the data x stored in the successor node is retrieved (line 9), and k becomes the new separator (line 10). Simultaneously, the consumer is notified with a some message that contains x as well as the continuation of the queue.

The code for Node, shown in the lower part of Listing 5, is fairly straightforward. The only remarkable feature of a node is that its state is a combination of DATA, containing the data stored in the node, and either one NEXT or one LAST message, depending on whether the node is linked to a subsequent node or not. In the former case, the NEXT message

carries a reference to the subsequent node. The operations `link` and `unlink` have the effect of switching these two states.

Note that `link` (line 17) and `unlink` (line 19) do not return the continuation of the node on which they are invoked (in the case of `unlink`, a node `n` is in fact returned, but that is the subsequent node in the chain, not the node itself). We have made this choice given that `Node` is only meant to be an auxiliary structure for implementing `Queue` and given that, according to the code for `Queue`, `link` and `unlink` are always the *last* operations invoked on a node by its owners.

Let us now devise types for queues and nodes, starting from the formers. The types t_{enq} and t_{deq} respectively expose operations for enqueueing and dequeueing data into/from the queue. The state of the queue is always the combination of `HEAD` and `TAIL`. Formally, we have

$$\begin{array}{rcl} t_{\mathsf{enq}} & = & \mathtt{enqueue}(\mathtt{int}, \mathtt{reply}(t_{\mathsf{enq}})) \\ t_{\mathsf{deq}} & = & \mathtt{dequeue}(\mathtt{none}(t_{\mathsf{deq}}) \oplus \mathtt{some}(\mathtt{int}, t_{\mathsf{deq}})) \\ t_{queue} & = & \mathtt{HEAD}(t_{\mathsf{ANY}}) \otimes \mathtt{TAIL}(\mathtt{link}(t_{\mathsf{ANY}})) \otimes t_{\mathsf{enq}} \otimes t_{\mathsf{deq}} \end{array}$$

Note that t_{enq} and t_{deq} in t_{queue} are combined by $\otimes$, meaning that they do *not* describe mutually exclusive behaviors. This is what allows two different threads, one producer and one consumer, to simultaneously access the same queue. The type t_{ANY} describes the interface of a `Node` whose state (either `NEXT` or `LAST`) is unknown, and will be detailed shortly. The arguments of `HEAD` and `TAIL` have different types, reflecting the fact that the queue uses them differently. In particular, nodes stored in `HEAD` can be in any state, hence we use the type t_{ANY} in $\mathtt{HEAD}(t_{\mathsf{ANY}})$, while nodes stored in `TAIL` do not have a successor and the only operation invoked on them from the queue is `link` (line 4), hence we use the type $\mathtt{link}(t_{\mathsf{ANY}})$ in $\mathtt{TAIL}(\mathtt{link}(t_{\mathsf{ANY}}))$.

Nodes expose three different public interfaces, depending on whether their state is `LAST`, `NEXT`, or uncertain. Correspondingly, we have three types t_{LAST}, t_{NEXT}, and t_{ANY} that satisfy the following equations:

$$\begin{array}{rcl} t_{\mathsf{LAST}} & = & \mathtt{link}(t_{\mathsf{ANY}}) \otimes t_{\mathsf{ANY}} \\ t_{\mathsf{NEXT}} & = & \mathtt{unlink}(\mathtt{reply}(t_{\mathsf{ANY}})) \oplus t_{\mathsf{ANY}} \\ t_{\mathsf{ANY}} & = & \mathtt{get\_data}(\mathtt{reply}(\mathtt{int}, t_{\mathsf{ANY}})) \oplus \mathtt{has\_next}(\mathtt{no}(t_{\mathsf{ANY}}) \oplus \mathtt{yes}(t_{\mathsf{NEXT}})) \end{array}$$

The operations `get_data` and `has_next` are always available, as witnessed by the fact that they occur in t_{ANY}. On the contrary, `link` is available only on nodes in state `LAST`, while `unlink` is available only on nodes in state `NEXT`. There is a subtle, but fundamental difference between t_{LAST} and t_{NEXT} in that `unlink` can be chosen *in alternative to* the operations in t_{ANY} (see the $\oplus$ in the equation for t_{NEXT}) whereas `link` must be performed *in addition to* one of the operations in t_{ANY} (see the $\otimes$ in the equation for t_{LAST}). The rationale for this asymmetry is due to the fact that queues can be simultaneously accessed by both a producer and a consumer. When a queue is empty there will be two threads acting on the only node in it, which is the separator: the producer will try to `link` the separator to a node that stores the data being enqueued (line 4) while the consumer will query the separator with `has_next` to check whether it is followed by another node (line 6). If `has_next` executes first and answers `no`, the type of the reference returned to the consumer is t_{ANY} and not t_{LAST}, for the capability of performing a `link` is exclusive to the producer. To allow simultaneous invocations of `link` and `get_data` we use $\otimes$ in the equation for t_{LAST}. In the code of `Queue`, such simultaneous access to the separator is best illustrated on line 12, where the `node` separator is stored in *both* `HEAD` and `TAIL`. On the contrary, `unlink` is always the last operation invoked on a node and is therefore mutually exclusive with `get_data` and `has_next`, which explains the $\oplus$ in the equation for t_{NEXT}.

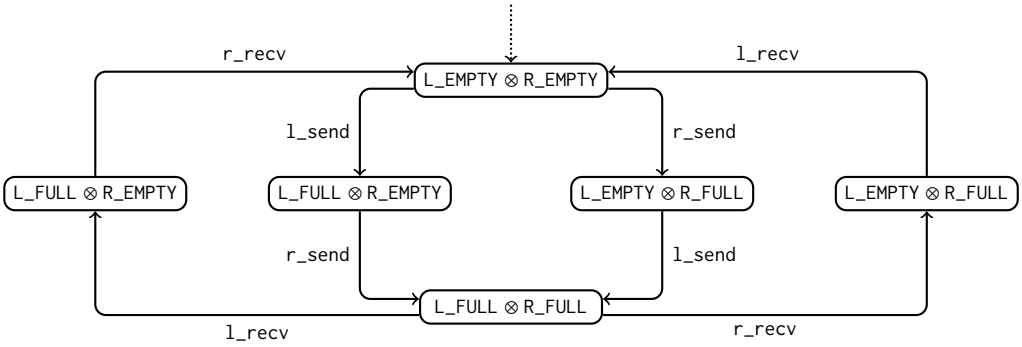

Fig. 2. Transition diagram of the full-duplex channel.

The type

$$t_{node} \stackrel{\text{def}}{=} \mathsf{DATA(int)} \otimes \big((\mathsf{LAST} \otimes t_{\mathsf{LAST}}) \oplus (\mathsf{NEXT}(t_{\mathsf{ANY}}) \otimes t_{\mathsf{NEXT}})\big)$$

describes the overall set of valid message configurations that can be sent to a Node.

7.3 Full-Duplex Channel

In the example of Section 7.2 we deliberately chose to have a compound state instead of a monolithic one LIST to enable the concurrent access to the object. In this section we discuss one final example where the compound state arises naturally as the combination of the states of two independent one-place buffers. The example we consider is that of a bidirectional, full-duplex channel for connecting two peer processes, called "left" and "right" and identified by a letter $p \in \{\mathtt{l}, \mathtt{r}\}$. *Full-duplex* communication allows the two peers to simultaneously cross-post messages on the channel, so as to maximize parallelism.

The channel is modeled in the Objective Join Calculus thus:

```
def Channel = new(r) ▷
  def o = L_EMPTY   | l_send(v,r) ▷ o.L_FULL(v) | r.reply(o)
       or L_FULL(v) | r_recv(r)   ▷ o.L_EMPTY   | r.reply(v,o)
       or R_EMPTY   | r_send(v,r) ▷ o.R_FULL(v) | r.reply(o)
       or R_FULL(v) | l_recv(r)   ▷ o.R_EMPTY   | r.reply(v,o)
  in o.L_EMPTY | o.R_EMPTY | r.reply(o)
in ···
```

The channel provides two pairs of operations p_send and p_recv used by peer p for sending and receiving messages. For the left peer, L_EMPTY represents the empty buffer and L_FULL(v) the full buffer with a value v. Tags R_EMPTY and R_FULL are used for representing the buffer of the right peer in a similar way. Observe that each buffer is either empty or full just as discussed in Section 7.1, but the two buffers coexist and can change state independently. This means that L_EMPTY and L_FULL are *or-states*, and so are R_EMPTY and R_FULL: on the contrary, all pairs shown in Figure 2 are *and-states*. This will be reflected in the type of the channel, where different states of the same buffer are combined by $\oplus$, whereas states of different buffers are combined by $\otimes$.

The code for `Channel` allows each peer to perform send and receive operations in any order. In practice, since the buffers are one-place, it makes sense to enforce a usage protocol such that each peer p alternates send and receive operations. In this way, the p_send of peer p fills the corresponding buffer and enables the $\overline{p}$_recv of the other peer $\overline{p}$, but only after $\overline{p}$ has sent its own message. Figure 2 depicts the transition diagram of the full-duplex channel used according to this protocol. The interface of the channel from the viewpoint of p is described by the type $t_{p\mathtt{s}}$ defined by

$$t_{p\mathtt{s}} = p\_\mathtt{send}(\mathtt{int}, \mathtt{reply}(t_{p\mathtt{r}}))$$
$$t_{p\mathtt{r}} = p\_\mathtt{recv}(\mathtt{reply}(\mathtt{int}, t_{p\mathtt{s}}))$$

The types of the interfaces are combined with state message types to form the type of the channel as follows

$$\begin{aligned} t_{chan} \stackrel{\text{def}}{=}\ & (\mathtt{L\_EMPTY} \otimes \mathtt{R\_EMPTY} \otimes t_{\mathtt{l}\mathtt{s}} \otimes t_{\mathtt{r}\mathtt{s}}) \oplus (\mathtt{L\_FULL} \otimes \mathtt{R\_FULL} \otimes t_{\mathtt{l}\mathtt{r}} \otimes t_{\mathtt{r}\mathtt{r}}) \\ & \oplus (\mathtt{L\_FULL} \otimes \mathtt{R\_EMPTY} \otimes t_{\mathtt{l}\mathtt{r}} \otimes t_{\mathtt{r}\mathtt{s}}) \oplus (\mathtt{L\_EMPTY} \otimes \mathtt{R\_FULL} \otimes t_{\mathtt{l}\mathtt{s}} \otimes t_{\mathtt{r}\mathtt{r}}) \\ & \oplus (\mathtt{L\_FULL} \otimes \mathtt{R\_EMPTY} \otimes t_{\mathtt{l}\mathtt{s}} \otimes t_{\mathtt{r}\mathtt{r}}) \oplus (\mathtt{L\_EMPTY} \otimes \mathtt{R\_FULL} \otimes t_{\mathtt{l}\mathtt{r}} \otimes t_{\mathtt{r}\mathtt{s}}) \end{aligned}$$

where we have elided the type of values in the buffers for readability.

Inspection of t_{chan} reveals that the reference `o` returned on line 6 has type $t_{\mathtt{l}\mathtt{s}} \otimes t_{\mathtt{r}\mathtt{s}}$, that is the composition of the two public interfaces of the channel, each corresponding to one of the peers. Therefore, the *same* channel object can be used by two parallel processes, according to these two types, as illustrated by the code snippet below:

```
let c = Channel.new() in      (* c : t_ls ⊗ t_rs *)
{ let c   = c.l_send(1) in    (* c : t_lr *)
  let v,c = c.l_recv in ...   (* c : t_ls *)
| let c   = c.r_send(2) in    (* c : t_rr *)
  let v,c = c.r_recv in ... } (* c : t_rs *)
```

The internal state of the full-duplex channel is the *combination* of distinct messages `L_`x and `R_`y that are consumed and produced *concurrently* by the users of the channel. In particular, each reaction rule in `Channel` changes only *part* of the channel's state, leaving the rest unchanged. The last side condition of rule [T-REACTION] verifies that such partial change maintains the channel's overall state in one of the configurations described by t_{chan}. The interested reader can verify that each reaction rule is indeed well typed with respect to t_{chan}.

As a final consideration, the fact that t_{chan} and the diagram in Figure 2 list 6 configurations (instead of the 4 corresponding to all possible combinations of `L_`x and `R_`y) suggests that the interface of the channel depends not only on its *current state* (encoded as a pair `L_`$x \otimes$ `R_`y) but also on its *past history*. For instance, in the two states identified by the combination of messages `L_FULL` $\otimes$ `R_EMPTY`, the peer `l` has produced its own message while the buffer of peer `r` is empty. But this can be either because `r` has not produced its own message yet, or because `r` has indeed produced the message, and peer `l` has already received it.

7.4 Programming Patterns for Concurrent TSOP in the Objective Join Calculus

By now we have illustrated a broad range of examples of concurrent objects with structured protocols and methods that (partially) update their state. Overall, we can classify methods according to three usage patterns.

(1) A method that can be invoked only when the receiver object is in a state for which the method is valid. In this case, aliasing of the object must be controlled (in our case, by giving the object a linear type) so that the type system can statically track its state. Examples of methods following this pattern are `release` for locks, `next`

for iterators, `insert` and `remove` for buffers (the single-threaded version), `link` and `unlink` for nodes of the queue.

(2) A method that can be invoked regardless of the state of the receiver object, relying on the reaction semantics of the Join Calculus to suspend the invocation until the object is in a state for which the method is valid. The method can (but need not) be given an unlimited type (using $*$) to permit object aliasing. Examples of methods following this pattern are `acquire` for locks, `insert` and `remove` for buffers (the multi-threaded versions), `l_recv` and `r_recv` for full-duplex channels.

(3) A method that can be invoked regardless of the state of the receiver object and notifies the caller with a message that discloses some information about the object's state. As a result of the invocation, the caller may have acquired enough information for performing subsequent state-specific invocations. Examples of methods following this pattern are `peek` for iterators and `has_next` for nodes of the queue.

8 IMPLEMENTATION ASPECTS

In this section we discuss the practical feasibility of our approach and we provide some evidence that it is amenable to be integrated in a practical programming language. TSOP involves design and implementation aspects covering both the *program level*, that is the runtime support for handling messages, matching join patterns, firing reactions, and the *type level*, namely the compile-time support that enforces our typing discipline. Concerning the program level and considering that our approach relies on a standard formulation of the (Objective) Join Calculus, we discuss two different strategies for adopting our programming model: the first one makes use of (existing) implementations of the Join Calculus (Section 8.1); the second one rests on the tight analogies between TSOP realized in the Objective Join Calculus and the Actor Model, a programming paradigm that also combines OOP and concurrency (Section 8.2). Concerning the type level, we only discuss some key issues concerning the implementation of the type discipline advocated in the paper (Section 8.3) and leave a more in-depth investigation to future work.

8.1 TSOP with Implementations of the Objective Join Calculus

There exist a number of standalone, embedded, and library implementations of the Join Calculus: native support for join patterns is provided in JoCaml (Fournet et al. 2003b), Join Java (Itzstein and Jasiunas 2003), and in Cω (Benton et al. 2004; Microsoft Research 2004), among others; library implementations of join patterns are available for C# (Russo 2007; Turon and Russo 2011), Visual Basic (Russo 2008), Scala (Haller and Van Cutsem 2008), Erlang (Plociniczak and Eisenbach 2010). Both native and library implementations of join patterns have pros and cons. Natively supported join patterns allow for specific optimizations (Le Fessant and Maranget 1998) and analysis techniques (Fournet et al. 1997; Patrignani et al. 2011), but they are currently available only for niche programming languages that enjoy limited popularity. Library implementations of join patterns are (or can be made) available for all mainstream programming languages and therefore integrate more easily with existing code and development environments, but they might be constrained by the syntax and typing discipline of the host language. Nonetheless, carefully crafted implementations can perform and scale remarkably well (Turon and Russo 2011).

Listing 6 presents a Scala implementation of the full-duplex channel (Section 7.3) using the Scala Joins library (Haller and Van Cutsem 2008), which provides a simple DSL for embedding Join-style definitions in Scala.[1] The code defining the `Channel` class has a straightforward correspondence with its formal counterpart. The class consists of a set of

[1]The example has been written and tested using a variant of Scala Joins 0.4 that has been patched to make it compatible with Scala 2.11.6.

```
class Channel[TL, TR] extends Joins {
  // MESSAGES //
  private object L_EMPTY extends NullaryAsyncEvent
  private object R_EMPTY extends NullaryAsyncEvent
  private object L_FULL extends AsyncEvent[TL]
  private object R_FULL extends AsyncEvent[TR]
  object l_send extends SyncEvent[Unit, TL]
  object r_send extends SyncEvent[Unit, TR]
  object l_recv extends NullarySyncEvent[TR]
  object r_recv extends NullarySyncEvent[TL]
  // REACTION RULES //
  join {
    case L_EMPTY() and l_send(v) => L_FULL(v)
                                    l_send reply {}
    case L_FULL(v) and r_recv()  => L_EMPTY()
                                    r_recv reply v
    case R_EMPTY() and r_send(v) => R_FULL(v)
                                    r_send reply {}
    case R_FULL(v) and l_recv()  => R_EMPTY()
                                    l_recv reply v
  }
  // INITIALIZATION //
  L_EMPTY()
  R_EMPTY()
}

class Process(chan : Channel[Int, String]) {
  def runRight(v : String) : Unit = {
    println("Right sends " + v); chan.r_send(v)
    println("Right receives " + chan.r_recv())
    runRight(v + "*") }
  def runLeft(v : Int) : Unit = {
    println("Left sends " + v); chan.l_send(v)
    println("Left receives " + chan.l_recv())
    runLeft(v + 1) }
}

object TestChannel extends App {
  val chan = new Channel[Int, String]
  Future { new Process(chan).runLeft(1930) }
  Future { new Process(chan).runRight("Pluto") }
  Thread.sleep(5000000)
}
```

Listing 6. The full-duplex channel in Scala Joins.

event declarations specifying the messages that can be targeted to instances of the class (lines 3–10), the *reaction rules* specifying the behavior of instances of the class (lines 12–21), as well as the initial state of each instance (lines 23–24). The main program (lines 39–41) creates a channel that is shared by two asynchronous processes that exchange integers and strings in full-duplex. The messages representing the state of the channel (lines 3–6) are `private` to enforce encapsulation of the state. The public interface (lines 7–10) is represented by synchronous events: invoking a `Channel`'s public operation (lines 29–30 and 33–34) suspends the calling thread until the matching reaction has fired and a result has been returned (lines 14,16,18,20). In the formal model, all message sends are asynchronous and sequentiality is encoded with explicit continuations (see the reference `c` in the user code for the channel in Section 7.3). The reaction rules that govern the channel behavior are defined by means of a call to the `join` method (inherited from the `Joins` superclass in the Scala Joins library) that takes as a parameter a partial function encoding the join patterns in terms of pattern matching. Pattern composition is then achieved by means of the `and` combinator, whose definition exploits Scala's extractors and extensible pattern matching. Note that `Channel` is parametric in the types `TL` and `TR` of the messages exchanged over the full-duplex channel. This possibility, not accounted for in the formal presentation of the type system (Section 5), comes for free thanks to Scala's support for generics.

8.2 TSOP with Actors

The Actor Model (Agha 1986; Hewitt et al. 1973) is a programming paradigm that blends OOP, concurrency, and message-passing. In this section we show that the actor programming model bears strong similarities with our approach to TSOP for concurrent objects. In particular, it is well known that actors can directly implement Finite State Machines: a machine's state corresponds to the actor's behavior, which specifies how the actor handles incoming messages. The ability of the actor to change its current behavior in response to some message, say `m`, corresponds to a state transition labelled with `m`. Unlike the chemical model that underlies the Join Calculus, however, where both states and operations are uniformly encoded as messages, in actor systems behaviors and (ordinary) messages belong to distinct categories.

To illustrate, let us consider the implementation of the one-place buffer from Section 7.1 using Scala Akka Actors, shown in Listing 7. Two features of Akka should be kept in mind when looking at this code. First, as in the Objective Join Calculus, communication is asynchronous and responses are communicated by exchanging explicit continuations. Second, the default message handling policy of Akka differs from that of the Objective Join Calculus. In Akka, messages that cannot be handled by the current behavior of an actor are discarded, whereas in the Objective Join Calculus they keep floating in the *chemical soup* until, if ever, they form a molecule that matches the left hand side of a reaction. In order to realize this policy in Akka, we rely on mix-in composition by means of a trait `Chemical` that modifies an actor so that any message of type `ProtocolMsg` that is not handled by the current behavior is collected (method `chemReact` in line 9) and resent to `self` whenever the actor enters a new state which might be able to handle it (method `chemBecome` in line 8). Note that the default policy of discarding unhandled messages is specific to Akka. Other implementations of the Actor Model, such as the one of Erlang, allow unhandled messages to accumulate in the actor's mailbox.

With this machinery in place, the definition of the buffer (lines 19–32) closely resembles the corresponding Join definition: the two reactions are represented by the two partial functions `EMPTY` and `FULL` and the initial behavior of the buffer is `EMPTY` (line 31). Protocol messages are encapsulated in a `BufferProtocol` object (lines 13–16) and are parameterized w.r.t. the type of the values exchanged in the protocol.[2] Notice that `reply` and `replyVal` messages, used by the buffer to answer its users, are not tagged with the `ProtocolMsg` trait since only the buffer's incoming messages

[2]Since the type parameters of messages are not related to the `Buffer`'s type parameter, the Scala compiler raises a warning for line 24 since it fails to match the type argument of the case pattern due to type erasure. On the other hand, defining the generic case classes inside the `Buffer[T]` class would

```
trait ProtocolMsg

trait Chemical extends Actor {
  private val soup : ArrayBuffer[ProtocolMsg] = new ArrayBuffer()
  private def check() = { soup.map(self ! _); soup.clear }
  private def keep : Receive = { case msg:ProtocolMsg => soup.append(m) }

  def chemBecome(newState : Receive) = { context.become(newState); check() }
  def chemReact(behave : Receive) : Receive = behave orElse keep
}

object BufferProtocol {
  case class insert[T](value : T, replyTo : ActorRef) extends ProtocolMsg
  case class remove(replyTo : ActorRef) extends ProtocolMsg
  case class reply(o : ActorRef)
  case class replyVal[T](v : T, o : ActorRef)
}

class Buffer[T] extends Actor with Chemical {
  import BufferProtocol._
  def continuation : ActorRef = self

  def EMPTY = chemReact {
      case insert(x : T, r) => chemBecome(FULL(x))
                               r ! reply(continuation)
  }
  def FULL(x : T) = chemReact {
      case remove(r) => chemBecome(EMPTY)
                        r ! replyVal(x, continuation)
  }
  def receive = EMPTY
}
```

Listing 7. Scala Akka implementation of the one-place buffer.

are kept in the chemical soup. A more fine-grained chemical semantics could use specific subtypes of `ProtocolMsg` to better identify which of the saved messages have to be resent to self depending on the new actor state. However, at this stage we avoid any consideration about typing.

Finally, observe that if the buffer is used in a single-threaded way, that is at any time it has just one user that alternates `insert` and `remove` operations, then there is no need to refine the default Akka semantics that concerns message handling. In this case, we can avoid mixing-in the `Chemical` trait, remove the call to `chemReact`, and use standard `context.become` instead of `chemBecome` method for switching behavior.

As a second example of TSOP in Akka, we illustrate the implementation of the full-duplex channel (Section 7.3). The challenge of this example comes from the fact that, according to the Actor Model, an actor can only handle one

generate a visibility problem in the user code since such code can only indirectly access the `Buffer` object through an `ActorRef` reference. A more precise account of the typing of this code is postponed to future work when studying the implementation of the associated behavioral type discipline.

```
1  class BufferSubObj[T] extends Buffer[T] {
2     override def continuation : ActorRef = context.parent
3  }
4
5  object ChannelProtocol {
6    case class l_send[TL](value : TL, replyTo : ActorRef) extends ProtocolMsg
7    case class r_send[TR](value : TR, replyTo : ActorRef) extends ProtocolMsg
8    case class l_recv(replyTo : ActorRef) extends ProtocolMsg
9    case class r_recv(replyTo : ActorRef) extends ProtocolMsg
10 }
11
12 class Channel[TL, TR] extends Actor{
13   import ChannelProtocol._
14   import BufferProtocol._
15
16   val left  = context.actorOf(Props(new BufferSubObj[TL]))
17   val right = context.actorOf(Props(new BufferSubObj[TR]))
18
19   def receive = {
20     case l_send(v : TL, r) => left forward insert(v, r)
21     case r_recv(r)         => left forward remove(r)
22     case r_send(v : TR, r) => right forward insert(v, r)
23     case l_recv(r)         => right forward remove(r)
24   }
25 }
```

Listing 8. The full-duplex channel in Scala Akka.

message at a time. Therefore, a naive implementation of the full-duplex channel along the same lines of the buffer would constraint its ability to handle concurrent operations requested by its users. In order to recover (at least partially) such concurrent behavior of the full-duplex channel, we structure its implementation as the bottom-up *and*-composition of two children actors, each corresponding to a one-place buffer. The full-duplex channel object then only acts as a forwarder that delegates incoming messages to the appropriate child buffer. Listing 8 shows the code of the `Channel` actor. Its behavior (lines 19–24) is fixed and consists of forwarding incoming messages to its children of type `Buffer` (lines 16–17). The *and-states* of the channel depicted in Figure 2 then correspond to the composition of the states of the two children, each of them dealing with only part of the channel's state, thus allowing the two buffers to be concurrently accessed by the two users.

The type of the two buffers, `BufferSubObj`, specializes the chemical buffer by overriding the definition of the continuation reference (lines 1–3). Indeed, after sending a message to the channel, say `l_send`, a user of the channel expects to receive a `reply` message carrying the *channel's continuation*, that is the reference to the channel to be used in the rest of the protocol. Since such a `reply` message is not sent back by the channel itself but by one of its buffers, the `BufferSubObj` object overrides the `continuation` field (see Listing 7) to the parent actor. Also notice that the `ChannelProtocol` object does not mention `reply` and `replyVal` messages, since these are sent directly by the buffers. If desired, additional encapsulation can be provided to hide `BufferProtocol`'s replies by means of additional `ChannelProtocol`'s replies.

```
1  object FullDuplex extends App {
2    import ChannelProtocol._
3    import BufferProtocol._
4
5    val s = ActorSystem()
6    val channel = s.actorOf(Props(new Channel[Int, String]), "channel")
7    val leftUser = s.actorOf(Props(new Actor{
8      channel ! l_send(1, self)
9      def receive = runLeft(1)
10     def runLeft(v : Int) : Receive  = {
11       case reply(o)       => o ! l_recv(self)
12       case replyVal(x, o) => o ! l_send(v + 1, self)
13                              context.become(runLeft(v + 1))
14     }
15   }))
16   val rightUser = s.actorOf(Props(new Actor{
17     channel ! r_send("*", self)
18     def receive = runRight("*")
19     def runRight(v : String) : Receive = {
20       case reply(o)       => o ! r_recv(self)
21       case replyVal(x, o) => o ! r_send(v + "*", self)
22                              context.become(runRight(v + "*"))
23    }
24   }))
25   Thread.sleep(2000)
26   s.shutdown()
27 }
```

Listing 9. Two actors using the full-duplex channel.

To conclude, Listing 9 shows the Akka version of the channel's client code that we wrote both in the Join Calculus (Section 7.3) and in Scala Joins (Listing 6). In line 6 a channel actor is created and shared among two user actors (lines 7–15 and 16–24) that exchange integers and strings in full-duplex mode. Each user starts by sending a message to the `channel` actor reference (lines 8 and 17) and waits for a `reply(o)` message carrying the continuation reference `o` that is used to send the next receive message (lines 11 and 20). According to the buffer's protocol, receive messages reply with `replyVal(x, o)` messages, that are then handled by the peer actors by recursively unfolding their behavior (lines 12–13 and 21–22).

8.3 Type Checking and TSOP for Concurrent Objects

Imposing the typing discipline that we have described in Section 5 is more challenging to put into practice since this requires the implementation of a substructural type system that makes use of unconventional behavioral connectives. When using the Scala Joins library, the programmer can only rely on the native Scala type checker, which can verify that programs comply with the *interface* of the objects they use, but not necessarily with their *protocol*. The compiler can detect if a message of the wrong type is sent to an object or if a message is sent to an object that does not expose that message in its interface, but it cannot verify whether messages are sent in a particular order, or if a program fulfils the

obligations with respect to the objects it uses. In the case of Scala Akka, the compiler provides even weaker guarantees: actor behaviors are defined as partial functions of type `Any => Unit` so the compiler allows to send any type of message to any actor. Basic typing support for TSOP with actors might be provided by the experimental module Akka-Typed that is available in Scala Akka's latest release 2.4. This module introduces typed actor references and static type checking to precisely guarantee that only messages of the expected type are sent to typed actors. Interestingly, in order to account for the possibility that actors dynamically change their behavior, the module puts forward a continuation-passing programming discipline that is very similar to that we used in Join Calculus and in the examples above. However, no linear property is checked by Akka-Typed type system.

Besides these approaches based on Scala, in more general terms we envision two ways of implementing the behavioral typing discipline advocated in this paper: the first one is to develop a TSOP-aware programming language, possibly integrated with one or more host languages, in the style of Plaid (Aldrich et al. 2009; Sunshine et al. 2011); the second one is to superimpose our type system to that of an existing programming language, augmented with a DSL for TSOP. The first approach would grant us complete control over the type checker and would make it possible to take full advantage of typing information, for example to minimize the amount and nature of runtime checks concerning typestates. A major downside is that the language would likely enjoy limited popularity. The second approach has been pursued in Mungo (Kouzapas et al. 2016), a Java front-end that implements a behavioral type system for Java objects exposing dynamically changing interfaces. The key idea in Mungo is to have a pre-processing phase that analyzes the code using a typestate-sensitive type checker. If this phase is passed, the program is handed over to the standard Java compiler. The stratification of this architecture could favor the integration of our framework with a wider range of programming languages and development environments.

9 RELATED WORK

Typestate-Oriented Programming

DeLine and Fähndrich (2004) represent class states as invariants describing predicates over fields. They support verification in the presence of inheritance and depend on a classification of references as not aliased or possibly aliased. This approach is refined by Bierhoff and Aldrich (2007) and by Aldrich et al. (2009) with a flexible access permission system that permits state changes even in the presence of aliasing. Shared access permissions have been investigated in a concurrent framework by Stork et al. (2009, 2014), but their integration with the typestate mechanism has not been considered in these works. In Plaid (Sunshine et al. 2011), the typestate of an object directly corresponds to its class, and that class can change dynamically. Plaid supports the major state modeling features of Statecharts: state hierarchy, *or-states*, and *and-states*, allowing states dimensions to change independently.

The foundations of Plaid and, in general, of TSOP are formally studied by Garcia et al. (2014) using a nominal object-oriented language with mutable state and a native notion of typestate change. The language is also equipped with a permission-based type system integrated with a gradual typing mechanism that combines static and dynamic checking. Progress and type preservation properties are formally proved. Our type system for TSOP is *structural* instead of *nominal* in the sense that object types are related by finding a correspondence between messages rather than between class/state names (Definition 4.2). As a consequence, the effect of a state-changing operation on the interface of an object is tracked implicitly by the type of exchanged continuations. Approaches based on nominal type systems, such as those used by DeLine and Fähndrich (2004) and Garcia et al. (2014), use annotations such as `[Closed » Open]` that mention the name of the state(s) of the object before and after the operation.

To the best of our knowledge, TSOP has been investigated in a concurrent setting only by Damiani et al. (2008) and marginally by Gay et al. (2010). Damiani et al. (2008) develop a type and effect system for a Java-like language to trace how the execution of a method changes the state of the receiver object. To forbid access to fields that are not available in the current object's state, only direct invocations of methods on `this` can change the state of the current object. Since each class method is synchronized, two concurrent threads cannot simultaneously execute in the same object. Our approach relaxes such restrictions. For instance, both the concurrent queue (Section 7.2) and the full-duplex channel (Section 7.3) can be used in true concurrency by two processes, and each process is statically guaranteed to comply with (its view of) the object protocol. Gay et al. (2010) study an integration of typestate and session types targeting distributed objects. The focus of their work is on the modularization of sessions across different methods rather than on typestates themselves. In fact, their work rests on the assumption that *non-uniform* objects (those whose interface changes with time) must be used linearly.

Possibly Concurrent Objects with Dynamic Interfaces

In the actor model (Agha 1986; Hewitt et al. 1973) messages received by objects are handled by an internal single-threaded control which can dynamically change its behaviour, thereby changing the object/actor's state. In SCOOP (Nienaltowski 2007; West et al. 2015) each object is associated with a handler thread and the client threads wishing to send requests to the object must explicitly register this desire by using the `separate` construct. Unlike actors, SCOOP's threads have more control over the order in which the receiver processes messages: the messages from a single separate block are processed in order, without any interleaving. In addition, SCOOP allows pre/postcondition reasoning in a concurrent setting: before executing a method, the executing processor waits until the precondition is satisfied while each postcondition clause is evaluated individually and asynchronously. Pre/post conditions are reminiscent of state-sensitive operations distinctive of TSOP.

Behavioral description of objects with dynamically changing interfaces, sometimes called *active* or *non-uniform* objects, have been studied for more than two decades. Nierstrasz (1993) proposes finite-state automata for describing object protocols and failure semantics for characterizing the subtyping relation. Type-theoretic approaches based on similar notions have been subsequently studied for various object calculi and type languages, with varying degrees of ensured properties (Najm et al. 1999; Puntigam 2001a,b; Puntigam and Peter 2001; Ravara and Vasconcelos 2000). Drossopoulou et al. (2001) propose an approach to the static verification of programs using objects with dynamic interfaces based on runtime object re-classification. Aside from the fact that none of these works is based on the Join Calculus or addresses TSOP explicitly, all of these type systems are biased towards the description of *operations* rather than state. In the approach of Najm et al. (1999) and in that of Ravara and Vasconcelos (2000), types are variants of process algebras built on actions standing for method invocations. Choices and "parallel" composition roughly correspond to our $\oplus$ and $\otimes$ operators, and sequential composition is used to express the dynamic change of an object's interface. We model this aspect using explicit continuation passing, along the lines of existing works on sessions (Dardha et al. 2012; Gay and Vasconcelos 2010). By contrast, the type systems of Puntigam (2001a,b) and that of Puntigam and Peter (2001) are based on the decoration of object types with *tokens* akin to typestates, in the sense that they can be used to represent in an abstract form the internal state of object that affects the available operations. Tokens themselves can be annotated in such a way so as to express obligations on the use of objects.

Castegren and Wrigstad (2016) propose a type system that integrates traits and capabilities for controlling data races in concurrent objects. They introduce two connectives $\oplus$ and $\otimes$ whose semantics remotely resembles that of our own

behavioral connectives. The purpose of such operators is to capture the potential interferences between different traits of an object rather than to describe its intended usage protocol.

Session Types and Continuations

Object protocols are a particular instance of *behavioral types*, namely types that prescribe the valid sequence of interactions that can be performed by, or may occur between, given processes. *Session types*, of which Hüttel et al. (2016) provide a comprehensive survey, are probably the best known example of behavioral types. There are both analogies and key differences between object protocols and session types that are worth pointing out. An object protocol describes the valid ways in which an object can be used by its clients, pretty much as a (binary) session type (Honda 1993) describes the valid sequences of operations that can be performed on a communication channel. However, a binary session is always used to connect exactly two communicating processes, whereas a concurrent object may be shared among, and simultaneously accessed by, several clients. For this reason, the notion of "duality", which is key in binary session type theories to relate the behaviors of the two peers of a session, makes no sense in our setting. *Multiparty sessions* (Honda et al. 2016) generalize binary sessions to several interacting participants. Multiparty session types differ from our object protocols in two main respects: first, they provide a description of the allowed interactions from a neutral viewpoint, whereas our protocols are always associated with a single object. Second, multiparty session types are enforced by *projecting* the interactions concerning every single participant, so that the behavior of a single participant with respect to its peers is always described in terms of a *sequence* of operations. Again, this contrasts with object protocols where the order of method invocations is underspecified. In any case, in session type theories there is always a close correspondence between the structure of the code that realizes a given protocol and the structure of the protocol itself. In contrast, our approach allows for a form of behavioral polymorphism whereby the same shared concurrent object can be given different (incompatible) protocols according to the intended usage (*cf.* the protocols for the buffer object in Section 7.1).

Without additional mechanisms in place, binary and multiparty session type theories can guarantee *deadlock freedom* only within a single session. In presence of multiple, interleaved sessions, it is necessary either to make additional assumptions on the network topology, which must be acyclic (Wadler 2014), or to use a richer type structure (Coppo et al. 2016; Padovani 2014) to detect mutual dependencies between sessions. Hüttel et al. (2016) provide a survery of these and related techniques, whose adaptation to our typing discipline is left for future work.

The use of explicit continuations for describing structured behaviors has been inspired by the encoding of session types into linear channel types (Dardha et al. 2012). Our type system uses essentially the same technique, except that continuations are objects instead of channels. Continuations are convenient also when types account for structured protocols, to describe the effect of functions on channels (Gay and Vasconcelos 2010). Hu and Yoshida (2016) use continuations for modeling session channels as a collection of Java classes, each corresponding to a specific state of the protocol.

Types for the Join Calculus

Our language of behavioral types is an original contribution of this paper and is also the first behavioral type theory for the Objective Join Calculus. Other type systems for the Join Calculus have been formally investigated by Fournet et al. (1997) and Patrignani et al. (2011) and for the Objective Join Calculus by Fournet et al. (2003a). Fournet et al. (1997) discuss an extension of ML-style polymorphism to the Join Calculus and address the issues that arise when polymorphic

channels are joined in the same pattern; Patrignani et al. (2011) present a typing discipline to reason on the scope within which channels are allowed to be used, so as enforce a form of encapsulation. These two works are based on the original formulation of the Join Calculus (Fournet and Gonthier 1996), where a join definition introduces a bunch of *channel names* all at once. In particular, objects are modeled indirectly as the set of the operations they support, each operation being represented by a distinct channel. Since types are associated with channels, it is then difficult if at all possible to describe and reason about the overall behavior of objects. The seminal paper on the Objective Join Calculus by Fournet et al. (2003a) also introduces a type system for establishing basic safety and privacy properties. In particular, it distinguishes between *public* and *private* messages, the latter ones being used for encoding the internal state of objects. Once again, privacy information is associated with the single messages and object types solely specify their interface, but not their protocol. In our type system, the separation between public and private messages stems from the fact that distinct references to the same object may be given different types possibly combined using $\otimes$.

To the best of our knowledge, the work by Calvert and Mycroft (2012) is the only one that describes a form of analysis on join patterns that takes object behaviors into account and therefore that is remotely related to our type system. However, Calvert and Mycroft (2012) propose a control-flow analysis rather than a type system and their aim is to optimize code generated by join definitions, for instance detecting that some channels (called *signals*) never escape the scope of the object in which they are defined and that their associated message queue always contains (at most) one message. Similar properties can also be inferred by inspecting the type associated with objects in our type system.

Parametric Properties

A different approach to checking that programs comply with the protocol of the objects they use is by means of monitoring techniques for parametric properties (Jin et al. 2011; Meredith et al. 2010). Such techniques are based on runtime verification, hence they cannot rule out protocol violations that do not manifest themselves during one particular execution. However, they can be used for verifying rather complex properties involving non-regular protocols (like the fact that a lock is released as many times as it is acquired) and multiple objects (like the fact that a collection does not change while it is being iterated upon). Our type system can use linearity to capture some non-regular protocols. For instance, it requires locks to be released as many times as they are acquired. However, it cannot express contextual properties (like the fact that the lock is released within the same method that has acquired it), nor properties involving multiple objects, since different objects have unrelated types.

10 CONCLUDING REMARKS

We have given evidence that the Objective Join Calculus is a natural model for TSOP. The choice of this particular model allowed us (1) to approach TSOP in a challenging setting involving concurrency, object sharing/aliasing, and partial/concurrent state updates; (2) to capture the characterizing facets of TSOP (state-sensitive operations, explicit state change, runtime state querying, object protocols, multidimensional state, aliasing control) with the support of a simple and elegant language of behavioral types equipped with intuitive semantics and subtyping (Section 4); (3) to devise a type system (Section 5) that statically guarantees valuable properties (Section 6) and includes a characterization of safe, concurrent state updates in terms of subtyping (see the side conditions of [T-REACTION]).

In this paper we focused on the theoretical foundations of the chemical approach to TSOP and only glimpsed at the key aspects that concern its practical realization (Section 8). While much of the current research efforts are directed to

exploring the implementation of our approach, we report a non-exhaustive list of extensions and future developments that we find particularly relevant or intriguing.

In our type system, fine-grained aliasing control is realized by the $\otimes$ connective: an object of type (equivalent to) $t \otimes s$ can (actually, must) be used according to both t and s, by possibly parallel processes. Uncontrolled aliasing requires using the exponential $*$. However, neither $\otimes$ nor $*$ express with sufficient precision some forms of aliasing/sharing of objects. It would be interesting to investigate whether and how our type language integrates with other forms of aliasing control (Bierhoff and Aldrich 2007; Fähndrich and DeLine 2002; Stork et al. 2009).

Efficient compilation techniques for join patterns (Le Fessant and Maranget 1998) rely on atomic operations and finite-state automata for tracking the presence messages with a given tag. Our type system paves the way to further optimizations: for example, t_{lock} says that, when the method `release` is invoked, the lock is *for sure* in state `BUSY`. In other words, the reaction involving `BUSY` and `release` can be triggered without requiring an actual synchronization and invocations to `release` compiled as ordinary method calls. The availability of precise information on the usage protocol of objects can help reducing the amount of locking, simplifying the mechanisms (automata) that detect triggered reactions, and improving the representation of objects with typestate.

We have designed and implemented a type checking algorithm for our type system (Crafa and Padovani 2017). The algorithm relies on explicit type annotations for object definitions and infers the type of all the free occurrences of object references. Full inference of object protocols has been investigated in a number of works (a detailed survey is given by de Caso et al. (2013)), some of which use specification languages inspired to regular expressions (Henzinger et al. 2005) as we do.

Acknowledgments. The authors are grateful to the anonymous referees for their insightful comments and suggestions.

REFERENCES

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*. ACM, 1015–1022.

Nels E. Beckman, Duri Kim, and Jonathan Aldrich. 2011. An Empirical Study of Object Protocols in the Wild. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, Vol. LNCS 6813. 2–26.

Nick Benton, Luca Cardelli, and Cédric Fournet. 2004. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems* 26, 5 (2004), 769–804.

Gérard Berry and Gérard Boudol. 1992. The Chemical Abstract Machine. *Theoretical Compututer Science* 96, 1 (1992), 217–248.

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. ACM, 301–320.

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *Journal of ACM* 11, 4 (1964), 481–494.

Peter Calvert and Alan Mycroft. 2012. Control Flow Analysis for the Join Calculus. In *Proceedings of 19th International Symposium on Static Analysis (SAS'12)*, Vol. LNCS 7460. Springer, 181–197.

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *Proceedings of 30th European Conference on Object-Oriented Programming (ECOOP'16)*, Vol. LIPIcs 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:26.

John Conway. 1971. *Regular Algebra and Finite Machines.* William Clowes & Sons Ltd.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science* 26 (2016), 238–302. Issue 2.

Bruno Courcelle. 1983. Fundamental Properties of Infinite Trees. *Theoretical Computer Science* 25 (1983), 95–169.

Silvia Crafa and Luca Padovani. 2015. The Chemical Approach to Typestate-Oriented Programming. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*, Vol. ACM SIGPLAN Notices 50. ACM, 917–934. Issue 10.

Silvia Crafa and Luca Padovani. 2017. Cobalt**Blue**. (2017). Retrieved March 2017 from http://www.di.unito.it/~padovani/Software/CobaltBlue/index.html

Ferruccio Damiani, Elena Giachino, Paola Giannini, and Sophia Drossopoulou. 2008. A type safe state abstraction for coordination in Java-like languages. *Acta Informatica* 45, 7-8 (2008), 479–536.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *Proceedings of the 14th symposium on Principles and Practice of Declarative Programming (PPDP'12)*. ACM, 139–150.

Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology* 22, 3 (2013), 25:1–25:46.

Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, Vol. LNCS 3086. Springer, 465–490.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2001. Fickle: Dynamic Object Re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, Vol. LNCS 2072. Springer, 130–149.

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming Language Design and Implementation (PLDI'02)*. ACM, 13–24.

Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-Calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM, 372–385.

Cédric Fournet and Georges Gonthier. 2000. The Join Calculus: A Language for Distributed Mobile Programming. In *International Summer School on Applied Semantics (APPSEM)*, Vol. LNCS 2395. Springer, 268–332.

Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. 1997. Implicit Typing à la ML for the Join-Calculus. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, Vol. LNCS 1243. Springer, 196–212.

Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. 2003a. Inheritance in the Join Calculus. *Journal of Logic and Algebraic Programming* 57, 1-2 (2003), 23–69.

Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. 2003b. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *Lecture Notes of the 4th International School on Advanced Functional Programming (AFP'03)*, Vol. LNCS 2638. Springer, 129–158.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems* 36, 4 (2014), 12.

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular Session Types for Distributed Object-Oriented Programming. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM, 299–312.

Philipp Haller and Tom Van Cutsem. 2008. Implementing Joins Using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08)*, Vol. LNCS 5052. Springer, 135–152.

David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive Interfaces. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 31–40.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*. William Kaufmann, 235–245.

Kohei Honda. 1993. Types for Dyadic Interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, Vol. LNCS 715. Springer, 509–523.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *Journal of ACM* 63, 1 (2016), 9.

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE'16)*, Vol. LNCS 9633. Springer, 401–418.

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. 2016. Foundations of Session Types and Behavioural Contracts. *Comput. Surveys* 49, 1 (2016), 3:1–3:36.

G. Stewart Von Itzstein and Mark Jasiunas. 2003. On Implementing High Level Concurrency in Java. In *Proceedings of the 8th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC'03)*, Vol. LNCS 2823. Springer, 151–165.

Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 415–424.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems* 21, 5 (1999), 914–947.

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP'16)*. ACM, 146–159.

Fabrice Le Fessant and Luc Maranget. 1998. Compiling Join-Patterns. *Electronic Notes in Theoretical Computer Science* 16, 3 (1998), 205–224.

Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. 2010. Efficient monitoring of parametric context-free patterns. *Automated Software Engineering* 17, 2 (2010), 149–180.

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*. ACM, 267–275.

Microsoft Research. 2004. Cω. (2004). Retrieved March 2017 from https://www.microsoft.com/en-us/research/project/comega/

Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. 1999. Guaranteeing Liveness in an Object Calculus through Behavioural Typing. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'99) and Protocol Specification, Testing and Verification (PSTV'99)*, Vol. 156. Kluwer, 203–221.

Piotr Nienaltowski. 2007. *Practical Framework for Contract-Based Concurrent Object-Oriented Programming.* Ph.D. Dissertation. ETH Zurich.

Oscar Nierstrasz. 1993. Regular Types for Active Objects. In *Proceedings of the 8th annual conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93).* ACM, 1–15.

Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proceedings of the Joint 23rd EACSL Annual Conference on Computer Science Logic and 29th Annual ACM/IEEE Symposium on Logic In Computer Science (CSL-LICS'14).* ACM, 72:1–72:10.

Marco Patrignani, Dave Clarke, and Davide Sangiorgi. 2011. Ownership Types for the Join Calculus. In *Proceedings of the Joint 13th IFIP WG 6.1 International Conference FMOODS and 30th IFIP WG 6.1 International Conference FORTE (FMOODS/FORTE'11)*, Vol. LNCS 6722. Springer, 289–303.

Hubert Plociniczak and Susan Eisenbach. 2010. JErlang: Erlang with Joins. In *Proceedings of the 12th International Conference on Coordination Models and Languages (COORDINATION'10)*, Vol. LNCS 6116. Springer, 61–75.

Franz Puntigam. 2001a. State Inference for Dynamically Changing Interfaces. *Computer Languages* 27, 4 (2001), 163–202.

Franz Puntigam. 2001b. Strong Types for Coordinating Active Objects. *Concurrency and Computation: Practice and Experience* 13, 4 (2001), 293–326.

Franz Puntigam and Christof Peter. 2001. Types for Active Objects with Static Deadlock Prevention. *Fundamenta Informaticae* 48, 4 (2001), 315–341.

António Ravara and Vasco T. Vasconcelos. 2000. Typing Non-uniform Concurrent Objects. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, Vol. LNCS 1877. Springer, 474–488.

Claudio V. Russo. 2007. The Joins Concurrency Library. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL'07)*, Vol. LNCS 4354. Springer, 260–274.

Claudio V. Russo. 2008. Join Patterns for Visual Basic. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08).* ACM, 53–72.

Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - A Theory of Mobile Processes.* Cambridge University Press.

Sven Stork, Paulo Marques, and Jonathan Aldrich. 2009. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceedings of the 24th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09).* ACM, 933–940.

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. 2014. Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Transactions on Programming Languages and Systems* 36, 1 (2014), 2:1–2:42.

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* 12, 1 (1986), 157–171.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-Class State Change in Plaid. In *Proceedings of the 26th ACM international conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'11).* ACM, 713–732.

Aaron Joseph Turon and Claudio V. Russo. 2011. Scalable join patterns. In *Proceedings of the 8th annual conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93).* ACM, 575–594.

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418.

Scott West, Sebastian Nanz, and Bertrand Meyer. 2015. Efficient and reasonable object-oriented concurrency. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15).* ACM, 273–274.

A PROOFS

In this appendix, it is convenient to generalize the definition of *substitution* given in the main body of the paper. In particular, a substitution σ is a map from names to object names that differs from the identity for a finite subset of variables. We write $\mathrm{dom}(\sigma)$ for the (finite) set of variables for which σ is *not* the identity.

A.1 Properties of Subtyping

Proposition A.1. *The following properties hold:*

(1) *If* $T \otimes \mathtt{M} \in [\![t]\!]$ *and* $\overline{\mathtt{M}}' = \overline{\mathtt{M}}$, *then* $T \in [\![t[\mathtt{M}']]\!]$.
(2) *If* $T \in [\![t[\mathtt{M}]]\!]$, *then there exists* $\mathtt{M}'$ *such that* $\overline{\mathtt{M}}' = \overline{\mathtt{M}}$ *and* $T \otimes \mathtt{M}' \in [\![t]\!]$.

Proof. Both items are proved by an easy induction on t. □

Proposition A.2. *If* $t \leqslant s$ *and* $\overline{T} = \overline{S}$, *then* $t[T] \leqslant s[S]$.

Proof. It is straightforward to see that Definition 5.4 is only affected by the signature of a molecule type and not by the type of the message arguments. Therefore, we can assume $T = S$ without loss of generality. We prove the result when T is a single message type $\mathtt{M}$, the general statement following by a simple induction on the size of the molecule type T. Let $\bigotimes_{i\in[1,n]} \mathtt{m}_i(\tilde{s}_i) \in [\![s[\mathtt{M}]]\!]$. From Proposition A.1(2) we deduce that there exist $\mathtt{m}_0$ and $\tilde{s}_0$ such that $\{\mathtt{m}_0\} = \overline{\mathtt{M}}$ and $\bigotimes_{i\in[0,n]} \mathtt{m}_i(\tilde{s}_i) \in [\![s]\!]$. From the hypothesis $t \leqslant s$ we know that there exists a family of $\tilde{t}_i$ such that $\bigotimes_{i\in[0,n]} \mathtt{m}_i(\tilde{t}_i) \in [\![t]\!]$ and $\tilde{s}_i \leqslant \tilde{t}_i$ for every $i \in [0,n]$. We conclude $\bigotimes_{i\in[1,n]} \mathtt{m}_i(\tilde{t}_i) \in [\![t[\mathtt{M}]]\!]$ by Proposition A.1(1). □

Proposition A.3. *If $T_1 \otimes T_2 \leqslant S_1 \otimes S_2$ and $\overline{T_i} = \overline{S_i}$ for $i = 1, 2$ and $\overline{T_1} \cap \overline{T_2} = \emptyset$, then $T_i \leqslant S_i$ for every $i = 1, 2$.*

Proof. Easy application of Definition 4.2. □

Lemma A.4. *If $t \downarrow T$ and $t \leqslant s \otimes S$ and $\mathsf{usable}(s)$ and $\overline{T} = \overline{S}$, then $T \leqslant S$.*

Proof. Let $R \in [\![s]\!]$. Such R exists from the hypothesis $\mathsf{usable}(s)$. Then $R \otimes S \in [\![s \otimes S]\!]$. From the hypothesis $t \leqslant s \otimes S$ we deduce that there exist R' and T' such that $R' \otimes T' \in [\![t]\!]$ and $R' \otimes T' \leqslant R \otimes S$ with $\overline{R \otimes S} = \overline{R' \otimes T'}$. Let be $R = \mathtt{M}_1 \otimes \cdots \otimes \mathtt{M}_k$ and $S = \mathtt{M}_{k+1} \otimes \cdots \otimes \mathtt{M}_n$. From $R' \otimes T' \leqslant R \otimes S$ and $\overline{R \otimes S} = \overline{R' \otimes T'}$, we know that $R' \otimes T' = \mathtt{M}'_1 \otimes \cdots \otimes \mathtt{M}'_n$ where $\mathtt{M}'_i \leqslant \mathtt{M}_i$ for every $i \in [1,n]$. Now, from the hypotheses $\overline{T} = \overline{S}$ and $t \downarrow T$, we deduce that $T' = T$ and $R' = \mathtt{M}'_1 \otimes \cdots \otimes \mathtt{M}'_k$ and $T = \mathtt{M}'_{k+1} \otimes \cdots \otimes \mathtt{M}'_n$. Therefore we conclude $T \leqslant S$ from $\mathtt{M}'_i \leqslant \mathtt{M}_i$ for every $i \in [k+1, n]$. □

A.2 Properties of Environment Subtyping and Combinations

Proposition A.5. *The following properties hold:*

(1) $\mathsf{dom}(\Gamma \otimes \Delta) = \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Delta)$;
(2) *If $\Gamma_1 \leqslant \Delta_1$ and $\Gamma_2 \leqslant \Delta_2$, then $\Gamma_1 \otimes \Gamma_2 \leqslant \Delta_1 \otimes \Delta_2$.*

Proof. Item (1) follows easily from Definition 5.1. We prove item (2) showing that $\Delta_1 \otimes \Delta_2$ and $\Gamma_1 \otimes \Gamma_2$ satisfy the conditions of Definition 5.2.

Concerning condition (1) of Definition 5.2, from the hypotheses $\Gamma_i \leqslant \Delta_i$ we deduce $\mathsf{dom}(\Delta_i) \subseteq \mathsf{dom}(\Gamma_i)$ for every $i = 1, 2$. Therefore, from item (1) we deduce $\mathsf{dom}(\Delta_1 \otimes \Delta_2) = \mathsf{dom}(\Delta_1) \cup \mathsf{dom}(\Delta_2) \subseteq \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) = \mathsf{dom}(\Gamma_1 \otimes \Gamma_2)$.

Concerning condition (2) of Definition 5.2, consider $u \in \mathsf{dom}(\Delta_1 \otimes \Delta_2)$. We have to consider several cases, but we only discuss one, the others being similar or simpler. Suppose $u \in (\mathsf{dom}(\Delta_1) \setminus \mathsf{dom}(\Delta_2)) \cap \mathsf{dom}(\Gamma_2)$. From the hypotheses $\Gamma_i \leqslant \Delta_i$ we deduce $\Gamma_1(u) \leqslant \Delta_1(u)$ and $\mathsf{nl}(\Gamma_2(u))$, that is $\Gamma_2(u) \leqslant \mathbb{1}$. We conclude $(\Gamma_1 \otimes \Gamma_2)(u) = \Gamma_1(u) \otimes \Gamma_2(u) \leqslant \Delta_1(u) \otimes \mathbb{1} \simeq \Delta_1(u) = (\Delta_1 \otimes \Delta_2)(u)$ using the pre-congruence of $\leqslant$.

Concerning condition (3) of Definition 5.2, consider $u \in \mathsf{dom}(\Gamma_1 \otimes \Gamma_2) \setminus \mathsf{dom}(\Delta_1 \otimes \Delta_2)$, that is $u \in (\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)) \setminus (\mathsf{dom}(\Delta_1) \cup \mathsf{dom}(\Delta_2))$. We discuss only one case, the others being analogous. Suppose $u \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2)$. From the hypothesis $\Gamma_1 \leqslant \Delta_1$ we deduce $\mathsf{nl}(\Gamma_1(u))$ and we conclude $\mathsf{nl}((\Gamma_1 \otimes \Gamma_2)(u))$ by observing that $(\Gamma_1 \otimes \Gamma_2)(u) = \Gamma_1(u)$. □

Definition A.6 (environment substitution). The application of σ to the environment Γ is the environment $\sigma\Gamma$ defined by the following environment combination (cf. Defintion 5.1): $\sigma\Gamma \stackrel{\text{def}}{=} \bigotimes_{u\in\mathsf{dom}(\Gamma)} \sigma(u) : \Gamma(u)$.

Proposition A.7. *The following properties hold:*

(1) $\mathsf{dom}(\sigma\Gamma) = \sigma(\mathsf{dom}(\Gamma))$;
(2) $\sigma(\Gamma \otimes \Delta) = \sigma\Gamma \otimes \sigma\Delta$;
(3) $\Gamma \leqslant \Delta$ *implies* $\sigma\Gamma \leqslant \sigma\Delta$;

Proof. Items (1) and (2) follow easily from Definition 5.1 and Definition A.6. We prove item (3) showing that $\sigma\Gamma$ and $\sigma\Delta$ satisfy the conditions of Definition 5.2.

Concerning condition (1), we observe that $\mathsf{dom}(\sigma\Delta) = \sigma(\mathsf{dom}(\Delta)) \subseteq \sigma(\mathsf{dom}(\Gamma)) = \mathsf{dom}(\sigma\Gamma)$ using item (1).

Concerning condition (3), consider $u \in \mathsf{dom}(\sigma\Gamma) \backslash \mathsf{dom}(\sigma\Delta)$. From Definition A.6 we have $(\sigma\Gamma)(u) = \bigotimes_{v \in \mathsf{dom}(\Gamma), \sigma(v)=u} \Gamma(v)$. From the hypothesis $u \notin \mathsf{dom}(\sigma\Delta)$ we deduce that for every $v \in \mathsf{dom}(\Gamma)$ such that $\sigma(v) = u$ we have $v \notin \mathsf{dom}(\Delta)$. Therefore, from the hypothesis $\Gamma \leqslant \Delta$, we deduce $\mathsf{nl}(\Gamma(v))$ for every $v \in \mathsf{dom}(\Gamma)$ such that $\sigma(v) = u$. That is to say, $\Gamma(v) \leqslant \mathbb{1}$ for every $v \in \mathsf{dom}(\Gamma)$ such that $\sigma(v) = u$. Now we derive $(\sigma\Gamma)(u) \leqslant \mathbb{1}$, namely $\mathsf{nl}((\sigma\Gamma)(u))$ which is what we wanted to prove.

Concerning condition (2), consider $u \in \mathsf{dom}(\sigma\Delta) \subseteq \mathsf{dom}(\sigma\Gamma)$. Now we derive

$$
\begin{array}{lcll}
(\sigma\Gamma)(u) & = & \bigotimes_{v \in \mathsf{dom}(\Gamma), \sigma(v)=u} \Gamma(v) & \text{by Definition A.6} \\
& = & \bigotimes_{v \in \mathsf{dom}(\Gamma) \backslash \mathsf{dom}(\Delta), \sigma(v)=u} \Gamma(v) \otimes \bigotimes_{v \in \mathsf{dom}(\Delta), \sigma(v)=u} \Gamma(v) & \text{since } \mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma) \\
& \leqslant & \bigotimes_{v \in \mathsf{dom}(\Delta), \sigma(v)=u} \Gamma(v) & \text{proof of condition (3)} \\
& \leqslant & \bigotimes_{v \in \mathsf{dom}(\Delta), \sigma(v)=u} \Delta(v) & \text{by hypothesis } \Gamma \leqslant \Delta \\
& = & (\sigma\Delta)(u) & \text{by Definition A.6}
\end{array}
$$

which concludes the proof. □

A.3 Proof of Theorem 6.1 (Subject Reduction)

Lemma A.8. *If* $\vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \rightharpoonup \mathcal{D}' \Vdash \mathcal{P}'$, *then* $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.

Proof. We reason by cases on the heating rule being applied. We omit the discussion of rule [join] since it is similar to that of [par].

[null] Then $\mathcal{D}' = \mathcal{D}$ and $\mathcal{P} = \mathcal{P}', \texttt{null}$. From [t-solution] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [t-processes] we deduce that there exist Δ_1 and Δ_2 such that $\Delta = \Delta_1 \otimes \Delta_2$ and $\Delta_1 \vdash \mathcal{P}'$ and $\Delta_2 \vdash \texttt{null}$. From [t-sub] and [t-null] we deduce $\mathsf{nl}(\Delta_2)$, hence $\Delta \leqslant \Delta_1$. We conclude by transitivity of $\leqslant$ with an application of [t-solution].

[def] Then $\mathcal{D}' = \mathcal{D}, a = C$ and $\mathcal{P} = \mathcal{P}'', \texttt{def}\ a = C\ \texttt{in}\ P$ and $\mathcal{P}' = \mathcal{P}'', P$ and $a \notin \mathsf{fn}(\mathcal{P}'')$. From [t-solution] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [t-processes] we deduce that there exist Δ_1 and Δ_2 such that $\Delta = \Delta_1 \otimes \Delta_2$ and $\Delta_1 \vdash \mathcal{P}''$ and $\Delta_2 \vdash \texttt{def}\ a = C\ \texttt{in}\ P$. From [t-sub] and [t-object] we deduce that there exists t and Δ_2' such that $\Delta_2 \leqslant \Delta_2'$ and $a : t \vdash C$ and $\Delta_2', a : t \vdash P$. Using the hypothesis $a \notin \mathsf{fn}(\mathcal{P}'')$ we can assume, without loss of generality, that $a \notin \mathsf{dom}(\Gamma)$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma, a : t$ and $\Delta' \stackrel{\text{def}}{=} \Delta, a : t$ and observe that $\Gamma' \leqslant \Delta'$. We conclude $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.

[par] Then $\mathcal{D}' = \mathcal{D}$ and $\mathcal{P} = \mathcal{P}'', P_1 \mid P_2$ and $\mathcal{P}' = \mathcal{P}'', P_1, P_2$. From [t-solution] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [t-processes] we deduce that there exist Δ_1 and Δ_2 such that $\Delta = \Delta_1 \otimes \Delta_2$ and $\Delta_1 \vdash \mathcal{P}''$ and $\Delta_2 \vdash P_1 \mid P_2$. From [t-sub] and [t-par] we deduce that there exist Δ_{21} and Δ_{22} such that $\Delta_2 \leqslant \Delta_{21} \otimes \Delta_{22}$ and $\Delta_{2i} \vdash P_i$ for $i = 1, 2$. We conclude with an application of [t-processes] and one of [t-solution]. □

Lemma A.9. *If* $\vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \rightharpoonup \mathcal{D}' \Vdash \mathcal{P}'$, *then* $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.

Proof. We reason by cases on the cooling rule being applied. We omit the discussion of rule [join] since it is similar to that of [par].

[NULL] Then $\mathcal{D}' = \mathcal{D}$ and $\mathcal{P}' = \mathcal{P}, \mathtt{null}$. The result is immediate as $\mathtt{null}$ is well typed in the empty environment.

[DEF] Then $\mathcal{D} = \mathcal{D}', a = C$ and $\mathcal{P} = \mathcal{P}'', P$ and $\mathcal{P}' = \mathcal{P}'', \mathtt{def}\ a = C\ \mathtt{in}\ P$ and $a \notin \mathsf{fn}(\mathcal{P}'')$. From [T-SOLUTION] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [T-DEFINITIONS] we deduce that $\Gamma = \Gamma', a : t$ and $a : t \vdash C$. From [T-PROCESSES] we deduce that there exist Δ_1 and Δ_2 such that $\Delta = \Delta_1 \otimes \Delta_2$ and $\Delta_1 \vdash \mathcal{P}''$ and $\Delta_2 \vdash P$. From the hypothesis $a \notin \mathsf{fn}(\mathcal{P}'')$ we can assume, without loss of generality, that $a \in \mathsf{dom}(\Delta_2) \setminus \mathsf{dom}(\Delta_1)$. That is, $\Delta_2 = \Delta_2', a : s$ where $t \leqslant s$. We conclude by [T-SUB] and [T-OBJECT].

[PAR] Then $\mathcal{D} = \mathcal{D}'$ and $\mathcal{P} = \mathcal{P}'', P_1, P_2$ and $\mathcal{P}' = \mathcal{P}'', P_1 \mid P_2$. From [T-SOLUTION] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [T-PROCESSES] we deduce that there exist Δ_1, Δ_2, and Δ_3 such that $\Delta_1 \vdash \mathcal{P}''$ and $\Delta_2 \vdash P_1$ and $\Delta_3 \vdash P_2$. By [T-PAR] we obtain $\Delta_2 \otimes \Delta_3 \vdash P_1 \mid P_2$. We conclude with one application of [T-PROCESSES] and one of [T-SOLUTION]. □

LEMMA A.10. *If $\Gamma \vdash J :: T$ and $\Delta \vdash \sigma J :: S$ and $T \leqslant S$, then $\Delta \leqslant \sigma\Gamma$.*

PROOF. By induction on J and by cases on its shape.

[$J = \mathtt{m}(\tilde{x})$] Then $\Gamma = \{x_i : t_i\}_{i \in [1,n]}$ and $T = \mathtt{m}(t_1, \ldots, t_n)$ and $\sigma J = \mathtt{m}(a_1, \ldots, a_n)$ and $S = \mathtt{m}(s_1, \ldots, s_n)$ and $\Delta = \bigotimes_{i \in [1,n]} a_i : s_i$. From the hypothesis $T \leqslant S$ we deduce $s_i \leqslant t_i$ for every $i \in [1, n]$. To prove $\Delta \leqslant \sigma\Gamma$, observe that $\mathsf{dom}(\sigma\Gamma) = \mathsf{dom}(\Delta)$ and consider $a \in \mathsf{dom}(\Delta)$. We deduce:

$$\begin{array}{lll} \Delta(a) = \bigotimes_{i \in [1,n], a_i = a} s_i & & \text{by definition of } \Delta \\ = \bigotimes_{i \in [1,n], a_i = a} t_i & & \text{by pre-congruence of } \leqslant \\ = \bigotimes_{i \in [1,n], \sigma(x_i) = a} t_i & & \text{by definition of } \sigma \\ = (\sigma\Gamma)(a) & & \text{by Definition A.6} \end{array}$$

[$J = J_1 \mid J_2$] Then $\Gamma = \Gamma_1, \Gamma_2$ and $T = T_1 \otimes T_2$ and $\Delta = \Delta_1 \otimes \Delta_2$ and $S = S_1 \otimes S_2$ and $\Gamma_i \vdash J_i :: T_i$ and $\Delta_i \vdash \sigma J_i :: S_i$ for every $i = 1, 2$. Since T_1 and T_2 have disjoint signatures, and so do S_1 and S_2, from the hypothesis $T \leqslant S$ and Proposition A.3 we deduce $T_i \leqslant S_i$ for every $i = 1, 2$. By induction hypothesis we deduce that $\Delta_i \leqslant \sigma\Gamma_i$ for every $i = 1, 2$. We conclude $\Delta = \Delta_1 \otimes \Delta_2 \leqslant \sigma\Gamma_1 \otimes \sigma\Gamma_2 = \sigma(\Gamma_1 \otimes \Gamma_2) = \sigma(\Gamma_1, \Gamma_2) = \sigma\Gamma$ by Proposition A.5(2) and Proposition A.7(2). □

LEMMA A.11. *If $\Gamma \vdash M :: T$, then $\sigma\Gamma \vdash \sigma M :: T$.*

PROOF. By induction on the derivation of $\Gamma \vdash M :: T$ and by cases on the last rule applied.

[T-MSG-M] Then $M = \mathtt{m}(u_1, \ldots, u_n)$ and $\Gamma = \bigotimes_{i \in [1,n]} u_i : t_i$ and $\mathsf{usable}(t_1, \ldots, t_n)$. By Proposition A.7(2) we deduce $\sigma\Gamma = \bigotimes_{i \in [1,n]} \sigma(u_i) : t_i$. We conclude with one application of [T-MSG-M] observing that $\sigma M = \mathtt{m}(\sigma(u_1), \ldots, \sigma(u_n))$.

[T-COMP-M] Then $M = M_1 \mid M_2$ and $\Gamma = \Gamma_1 \otimes \Gamma_2$ and $T = T_1 \otimes T_2$ and $\Gamma_i \vdash M_i :: T_i$ for every $i = 1, 2$. By induction hypothesis we deduce $\sigma\Gamma_i \vdash \sigma M_i :: T_i$. We conclude with one application of [T-COMP-M] observing that $\sigma M = \sigma M_1 \mid \sigma M_2$ and by Proposition A.7(2). □

LEMMA A.12. *If $\Gamma \vdash P$, then $\sigma\Gamma \vdash \sigma P$.*

PROOF. By induction on the derivation of $\Gamma \vdash P$ and by cases on the last rule applied. We omit the discussion of [T-NULL] which is trivial.

[T-SEND] Then $P = u.M$ and $\Gamma = \Delta \otimes u : T$ and $\Delta \vdash M :: T$. By Lemma A.11 we deduce $\sigma\Delta \vdash \sigma M :: T$. By [T-SEND] we derive $\sigma\Delta \otimes \sigma(u) : T \vdash \sigma(u).\sigma M$. We conclude by observing that $\sigma\Gamma = \sigma(\Delta \otimes u : T) = \sigma\Delta \otimes \sigma(u) : T$ by Proposition A.7(2).

[T-PAR] Then $P = P_1 \mid P_2$ and $\Gamma = \Gamma_1 \otimes \Gamma_2$ and $\Gamma_i \vdash P_i$ for every $i = 1, 2$. By induction hypothesis we deduce $\sigma\Gamma_i \vdash \sigma P_i$ for every $i = 1, 2$. We conclude by Proposition A.7(2) and one application of [T-PAR], observing that $\sigma P = \sigma P_1 \mid \sigma P_2$.

[T-OBJECT] Then $P = \texttt{def}\ a = C\ \texttt{in}\ Q$ and there exists t such that $a : t \vdash C$ and $\Gamma, a : t \vdash Q$. Without loss of generality we may assume that no variable in $\mathrm{dom}(\Gamma)$ is mapped to a by σ. If this is not the case, we can suitably rename a in P where it is bound. Now we have $\sigma(\Gamma, a : t) = \sigma\Gamma, a : t$. By induction hypothesis we deduce $\sigma(\Gamma, a : t) \vdash \sigma Q$. We conclude with one application of [T-OBJECT].

[T-SUB] Then there exists Δ such that $\Gamma \leqslant \Delta$ and $\Delta \vdash P$. By induction hypothesis we deduce $\sigma\Delta \vdash \sigma P$. We conclude by Proposition A.7(3) and one application of [T-SUB]. □

Lemma A.13. *If* $\vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \to \mathcal{D} \Vdash \mathcal{P}'$, *then* $\vdash \mathcal{D} \Vdash \mathcal{P}'$.

Proof. We have $\mathcal{D} = \mathcal{D}', a = \{J_i \triangleright P_i\}_{i \in I}$ and $\mathcal{P} = \mathcal{P}'', a.\sigma J_k$ and $\mathcal{P}' = \mathcal{P}'', \sigma P_k$ for some $k \in I$. From [T-SOLUTION] we deduce that there exist Γ and Δ such that $\Gamma \leqslant \Delta$ and $\Gamma \vdash \mathcal{D}$ and $\Delta \vdash \mathcal{P}$. From [T-DEFINITIONS] we deduce that there exist t such that $\Gamma(a) = t$ and $a : t \vdash \{J_i \triangleright P_i\}_{i \in I}$. From [T-CLASS] and [T-REACTION] we deduce that there exist Γ_0, T, and s such that (1) $\Gamma_0 \vdash J_k :: T$ and (2) $\Gamma_0, a : s \vdash P_k$ and furthermore (3) $t \downarrow T$, and (4) $t \leqslant t[T] \otimes s$. From [T-PROCESSES] we deduce that there exist Δ_1, Δ_2, t_1, and t_2 such that $t \leqslant t_1 \otimes t_2$ and $\Delta = (\Delta_1, a : t_1) \otimes (\Delta_2, a : t_2)$ and $\Delta_1, a : t_1 \vdash \mathcal{P}''$ and $\Delta_2, a : t_2 \vdash a.\sigma J_k$ (if $a \notin \mathrm{fn}(\mathcal{P}'')$ we can take $t_1 = \mathbb{1}$). From [T-SUB] and [T-SEND] we deduce that there exist Δ_2', t_3, and S such that $\Delta_2 \leqslant \Delta_2'$ and $t_2 \leqslant t_3 \otimes S$ and (5) $\Delta_2', a : t_3 \vdash \sigma J_k :: S$. Overall, we have (6) $t \leqslant t_1 \otimes t_3 \otimes S$. Furthermore, it must be the case that $\overline{T} = \overline{S}$ for T and S are molecule types for a pattern and a molecule having the same signature. From (3), (6) and $\mathrm{usable}(t_1 \otimes t_3)$, which comes by the following Lemma A.15, by an application of Lemma A.4 we deduce $T \leqslant S$. From (1), (5), and Lemma A.10, we deduce that $\Delta_2', a : t_3 \leqslant \sigma\Gamma_0$. From Proposition A.7(3) we derive $\Delta_2', a : t_3 \otimes s \leqslant \sigma\Gamma_0 \otimes a : s$. From (2), Lemma A.12, and one application of [T-SUB] we derive $\Delta_2, a : t_3 \otimes s \vdash \sigma P_k$ and from this we can further derive $(\Delta_1 \otimes \Delta_2), a : t_1 \otimes t_3 \otimes s \vdash \mathcal{P}'', \sigma P_k$ with an application of [T-PROCESSES]. From (6) and Proposition A.2 we deduce $t[T] \leqslant (t_1 \otimes t_3 \otimes S)[S] = ((t_1 \otimes t_3)[S] \otimes S) \oplus (t_1 \otimes t_3 \otimes S[S]) \leqslant t_1 \otimes t_3$. We conclude $t \leqslant t_1 \otimes t_3 \otimes s$ using (4) and pre-congruence of $\leqslant$. □

Theorem 6.1 is now obtained as the combination of Lemmas A.8, A.9 and A.13.

A.4 Proof of Theorem 6.2 (Respected Prohibitions)

Lemma A.14. $\Gamma \vdash M :: T$ *implies* $\mathrm{usable}(\Gamma)$.

Proof. By induction on the derivation of $\Gamma \vdash M :: T$ and by cases on the last rule applied.

[T-MSG-M] Then $M = \mathtt{m}(u_1, \ldots, u_n)$ and $\Gamma = \bigotimes_{i=1..n} u_i : t_i$ where $\mathrm{usable}(t_1, \ldots, t_n)$. The result follows immediately.

[T-COMP-M] Then $\Gamma = \Gamma_1 \otimes \Gamma_2$ and $M = M_1 \mid M_2$ and $T = T_1 \otimes T_2$ and $\Gamma_i \vdash M_i :: T_i$ for $i = 1, 2$. By induction hypothesis we deduce $\mathrm{usable}(\Gamma_i)$ for every $i = 1, 2$. We conclude $\mathrm{usable}(\Gamma)$ since $\mathrm{usable}(s_1)$ and $\mathrm{usable}(s_2)$ imply $\mathrm{usable}(s_1 \otimes s_2)$. □

The following Lemma is an equivalent reformulation of Lemma 6.6.

Lemma A.15. $\Gamma \vdash P$ *implies* $\mathrm{usable}(\Gamma)$.

Proof. By induction on the derivation of $\Gamma \vdash P$ and by cases on the last rule applied. Most cases are simple or they follow immediately from the induction hypothesis. The case of [T-SEND] is a consequence of Lemma A.14. If the last rule applied is [T-SUB], then $\Delta \vdash P$ where $\Gamma \leqslant \Delta$. By induction hypothesis we deduce usable(Δ). We conclude usable(Γ) by Definition 5.2 since $\mathsf{nl}(t)$ implies usable(t). □

Theorem A.16 (Theorem 6.2). *If* $\Gamma, a : t \vdash \mathcal{P}, a.\mathtt{m}_1(\tilde{c}_1), \ldots, a.\mathtt{m}_n(\tilde{c}_n)$, *then there exist* S *and* $\tilde{s}_i$ *such that* $t \leqslant S \otimes \bigotimes_{i\in[1,n]} \mathtt{m}_i(\tilde{s}_i)$.

Proof. From the hypothesis and [T-PROCESSES] we deduce that there exist Γ_i and t_i for $i \in [0,n]$ such that $\Gamma = \bigotimes_{i\in[0,n]} \Gamma_i$ and $t = \bigotimes_{i\in[0,n]} t_i$ and $\Gamma_0, a : t_0 \vdash \mathcal{P}$ and $\Gamma_i, a : t_i \vdash a.\mathtt{m}_i(\tilde{c}_i)$ for every $i \in [1,n]$. If $a \notin \mathsf{fn}(\mathcal{P})$, we can take $t_0 = \mathbb{1}$ without loss of generality. From [T-SUB] and [T-SEND] we deduce that, for every $i \in [1,n]$, there exist $\tilde{s}_i$ and $\mathtt{m}_i$ such that $t_i \leqslant \mathtt{m}_i(\tilde{s}_i)$. From Lemma A.15 we deduce usable(t_0), namely there exist $S \in [\![t_0]\!]$. We conclude $t = \bigotimes_{i\in[0,n]} t_i \leqslant S \otimes \bigotimes_{i\in[1,n]} \mathtt{m}_i(\tilde{s}_i)$ by definition of t and pre-congruence of $\leqslant$. □

Note that there is no molecule type S such that $S \simeq \mathbb{0}$, hence the property in Theorem 6.2 cannot be satisfied trivially.

A.5 Proof of Theorem 6.3 (Weakly Fulfilled Obligations)

Lemma A.17. *If* $\Gamma \vdash P$ *and* $a \in \mathsf{dom}(\Gamma) \setminus \mathsf{fn}(P)$, *then* $\mathsf{nl}(\Gamma(a))$.

Proof. By induction on the derivation of $\Gamma \vdash P$ and by cases on the last rule applied. We omit the discussion of [T-NULL] which is trivial.

[T-SEND] Then $\Gamma = \Gamma' \otimes u : T$ and $P = u.M$ and $\Gamma' \vdash M :: T$. An easy induction on the derivation of $\Gamma' \vdash M :: T$ suffices to establish that $\mathsf{dom}(\Gamma') = \mathsf{fn}(M)$, therefore $\mathsf{fn}(P) = \{u\} \cup \mathsf{fn}(M) = \mathsf{fn}(\Gamma)$. The statement holds vacuously since $\mathsf{dom}(\Gamma) \setminus \mathsf{fn}(P) = \emptyset$.

[T-PAR] Then $\Gamma = \Gamma_1 \otimes \Gamma_2$ and $P = P_1 \mid P_2$ and $\Gamma_i \vdash P_i$ for every $i = 1, 2$. By induction hypothesis, $a \in \mathsf{dom}(\Gamma_i) \setminus \mathsf{fn}(P_i)$ implies $\mathsf{nl}(\Gamma_i(a))$ for every $i = 1, 2$. We conclude by observing that $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ and $\mathsf{fn}(P) = \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2)$.

[T-OBJECT] Then $P = \mathtt{def}\ c = C\ \mathtt{in}\ Q$ and $\Gamma, c : t \vdash Q$ where, without loss of generality, we may assume $a \neq c$. By induction hypothesis we deduce that $a \in (\mathsf{dom}(\Gamma) \cup \{c\}) \setminus \mathsf{fn}(Q)$ implies $\mathsf{nl}(\Gamma(a))$. We conclude by observing that $\mathsf{fn}(P) = \mathsf{fn}(Q) \setminus \{c\}$.

[T-SUB] Then $\Delta \vdash P$ for some Δ such that $\Gamma \leqslant \Delta$. Recall that $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma)$ by Definition 5.2. Let $a \in \mathsf{dom}(\Gamma) \setminus \mathsf{fn}(P)$. We distinguish two sub-cases: if $a \in \mathsf{dom}(\Delta)$, then we conclude using the induction hypothesis; if $a \in \mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Delta)$, then $\mathsf{nl}(\Gamma(a))$ by Definition 5.2. □

Theorem 6.3 is just a reformulation of Lemma A.17.

Received March 2016; revised November 2016; accepted March 2017