Using π -Calculus Names as Locks

Daniel Hirschkoff

ENS de Lyon

daniel.hirschkoff@ens-lyon.fr

Enguerrand Prebet
Karlsruhe Institute of Technology
enguerrand.prebet@kit.edu

Locks are a classic data structure for concurrent programming. We introduce a type system to ensure that names of the asynchronous π -calculus are used as locks. Our calculus also features a construct to deallocate a lock once we know that it will never be acquired again. Typability guarantees two properties: deadlock-freedom, that is, no acquire operation on a lock waits forever; and leak-freedom, that is, all locks are eventually deallocated.

We leverage the simplicity of our typing discipline to study the induced typed behavioural equivalence. After defining barbed equivalence, we introduce a sound labelled bisimulation, which makes it possible to establish equivalence between programs that manipulate and deallocate locks.

1 Introduction

The π -calculus is an expressive process calculus based on the notion of name, in which name-passing is the primitive notion of interaction between processes. Processes of the π -calculus have been used to represent several aspects of programming, like data structures, protocols, or constructs such as functions, continuations, objects, and references. The π -calculus also comes with a well-developed theory of behavioural equivalence. This theory can be exploited to reason about contextual equivalence in programming languages, by translating programs as π -calculus processes.

In this work, we follow this path for locks, a basic data structure for concurrent programming. We study how π -calculus names can be used to represent locks. We show that the corresponding programming discipline in the π -calculus induces a notion of behavioural equivalence between processes, which can be used to reason about processes manipulating locks. This approach has been followed to analyse several disciplines for the usage of π -calculus names: linearity [15], receptiveness [25], locality [16], internal mobility [24], functions [23, 5], references [7, 21].

It is natural to represent locks in $A\pi$, the asynchronous version of the π -calculus [1, 9]. A lock is referred to using a π -calculus name. It is represented as an asynchronous output: the release of the lock. Dually, an input represents the acquire operation on some lock.

In this paper, we introduce $\pi\ell$ w, a version of the asynchronous π -calculus with only lock names. Two properties should be ensured for names to be used as locks: first, a lock can appear at most once in released form. Second, acquiring a lock entails the obligation to release it. For instance, process $\ell_1(x)$. $(\overline{\ell_1}\langle x\rangle \mid \overline{\ell_2}\langle x\rangle)$ has these properties: the process acquires lock ℓ_1 , then releases it, together with lock ℓ_2 . We remark that this this process owns lock ℓ_2 , which is released after ℓ_1 is acquired. We show that a simple type system can be defined to guarantee the two properties mentioned above.

When manipulating locks, it is essential to avoid the program from getting stuck in a state where a lock needs to be acquired but cannot be released. Consider the following process:

$$P_{\mathsf{dl}} \stackrel{\text{def}}{=} \ell_1(x). (\overline{\ell_1}\langle x \rangle \mid \overline{\ell_2}\langle x \rangle) \mid \ell_2(y). (\overline{\ell_1}\langle y \rangle \mid \overline{\ell_2}\langle y \rangle).$$

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).

EPTCS 387, 2023, pp. 76–96, doi:10.4204/EPTCS.387.7

© D. Hirschkoff and E. Prebet This work is licensed under the Creative Commons Attribution License. The subprocess on the left needs to acquire lock ℓ_1 , which is owned by the other subprocess, and symmetrically: this is a deadlock. Our type system rules out processes that exhibit this kind of cyclic dependency between locks. This is achieved by controlling parallel composition: two processes in parallel can share at most one lock name. Process P_{dl} thus cannot be typed, because names ℓ_1 and ℓ_2 are shared between the two subprocesses. The acyclicity property enjoyed by typable processes yields deadlock-freedom.

To avoid situations where a lock is in released state and cannot be accessed, $\pi \ell w$ also features a construct to deallocate a lock, called *wait*, inspired from [12]. Process $\ell((x))$. P waits until no acquire is pending on lock ℓ , at which point it deallocates ℓ , reading the final value stored in ℓ as x. The reduction rule for wait is

$$(\nu\ell)(\overline{\ell}\langle\nu\rangle \mid \ell((x)).P) \to P\{\nu/x\} \tag{1}$$

provided ℓ is not among the free names of P. In the reduction above, the restriction on ℓ disappears after the last interaction involving ℓ has taken place.

The main contributions of this work are the following:

- We introduce $\pi \ell w$, a π -calculus with higher-order locks (in the sense that locks can be stored in locks) and lock deallocation. The type system for $\pi \ell w$ controls the usage and the sharing of lock names between processes. We provide some examples to illustrate how locks can be manipulated according to the programming discipline induced by types.
- We show that typable processes in $\pi \ell w$ enjoy deadlock- and leak-freedom. The proofs rely on simple arguments involving the graph induced by the sharing of locks among processes.
- We analyse typed behavioural equivalence in $\pi \ell w$. Types restrict the set of contexts that can interact with processes, yielding a coarser behavioural equivalence than in the untyped case.
 - We first introduce typed barbed equivalence, written \simeq_w . Relation \simeq_w is defined by observing the behaviour of processes when they are placed in typable contexts. We then express the interactions between typed processes and typed context by means of a Labelled Transition System (LTS) that takes into account typing constraints. This allows us to introduce *typed bisimilarity*, \approx_w , the main proof technique to establish barbed equivalence: we indeed prove soundness, that is, $\approx_w \subseteq \simeq_w$.

We discuss several examples that help to understand how we can reason about behavioural equivalence in $\pi \ell w$. We are not aware of existing labelled equivalences taking into account name deallocation in the π -calculus.

Beyond $\pi \ell w$, we believe that \approx_w can be used as a building block when reasoning in the π -calculus about programs that use various features, among which locks.

The aforementioned contributions are presented in two steps. We first introduce $\pi\ell$, an asynchronous π -calculus with higher-order locks. $\pi\ell$ is obtained by adding the wait construct to $\pi\ell$. Several important ideas can be presented in $\pi\ell$, and we can build on the notions introduced for $\pi\ell$ to extend them for $\pi\ell$ w.

We now highlight some of the technical aspects involved in our work.

The type system for $\pi\ell$ guarantees deadlock-freedom, in the sense that for typable processes, an acquire operation cannot be blocked forever. This holds for *complete processes*: a process is complete if for every lock ℓ it uses, a release of ℓ is available. Availability need not be immediate, for instance the release operation on lock ℓ may be blocked by an acquire on ℓ' . We prove progress based on the fact that the type system guarantees acyclicity of the dependence relation between locks. Progress entails deadlock-freedom.

When adding the wait construct, we rely on a similar reasoning to prove leak-freedom for $\pi \ell w$, which in our setting means that all locks are eventually deallocated. The type system for $\pi \ell w$ is richer than the

one for $\pi\ell$ not only because it takes wait into account, but also because it makes it possible to transmit the obligation of releasing or deallocating a lock via another lock. For instance, it is possible, depending on the type of ℓ , that in process $\ell(\ell').P$, the continuation P has the obligation not only to release lock ℓ , but also to deallocate ℓ' , or release ℓ' , or both.

To define typed barbed equivalence in $\pi\ell$, written \simeq , we must take into account deadlock-freedom, which has several consequences. First, we observe complete processes: intuitively, computations in $\pi\ell$ make sense only for such processes, and a context interacting with a process should not be able to block a computation by never performing some release operation. Second, all barbs are always observable in $\pi\ell$. In other words, if ℓ is a free name of a complete typable process P, then P can never loose the ability to release ℓ . This is in contrast with barbed equivalence in the π -calculus, or in CCS, where the absence of a barb can be used to observe behaviours. We therefore adopt a stronger notion of barb, where the value stored in a lock, and not only the name of the lock, can be observed.

The ideas behind \simeq are used to define \simeq_w , typed barbed equivalence in $\pi \ell w$. A challenge when defining typed bisimilarity in $\pi \ell w$ is to come up with labelled transitions corresponding to the reduction in (1). Intuitively, if $P \xrightarrow{\ell((v))} P'$ (P deallocates ℓ and continues as P'), we must make sure that this transition is the last interaction at ℓ . We define a typed LTS to handle name deallocation, and show that bisimilarity is sound for barbed equivalence in $\pi \ell w$.

Paper outline. We study $\pi\ell$ in Section 2. We first expose the essential ideas of our deadlock-freedom proof in $CCS\ell$, a simple version of the Calculus of Communicating Systems [18] with lock names. After extending these results to $\pi\ell$, we define barbed equivalence for $\pi\ell$, written \simeq . We provide a labelled semantics that is sound for \simeq , and present several examples of behavioural equivalences in $\pi\ell$. In Section 3, we add the wait construct, yielding $\pi\ell$ w. We show how to derive leak-freedom, and define a labelled semantics, building on the ideas of Section 2. We discuss related and future work in Section 4.

2 $\pi \ell$, a Deadlock-Free Asynchronous π -Calculus

We present deadlock-freedom in the simple setting of CCS ℓ in Section 2.1. This approach is extended to handle higher-order locks in $\pi\ell$ (Section 2.2). We study behavioural equivalence in $\pi\ell$ in Section 2.3.

2.1 CCS\(\ell\): Ensuring Deadlock-Freedom using Composition

CCS ℓ is a simplification of $\pi\ell$, to present the ideas underlying the type system and the proof of deadlock-freedom. CCS ℓ is defined as an asynchronous version of CCS with acquire and release operations. We postulate the existence of an infinite set of *lock names*, written $\ell, \ell', \ell_1, \ldots$, which we often simply call names. CCS ℓ processes are defined by the following grammar:

$$P ::= \ell.P \mid \overline{\ell} \mid (\nu\ell)P \mid P_1 \mid P_2.$$

 ℓ is the release of lock ℓ . Process ℓ . P acquires ℓ and then acts as P—we say that P performs an *acquire* on ℓ . There is no $\mathbf{0}$ process in $CCS\ell$, intuitively because we do not take into consideration processes with no lock at all. Restriction is a binder, and we write fln(P) for the set of free lock names in P. If $\mathbb{S} = \{\ell_1, \dots, \ell_k\}$ is a set of lock names, we write $(v\mathbb{S})P$ for $(v\ell_1)\dots(v\ell_k)P$.

The definition of structural congruence, written \equiv , and reduction, written \rightarrow , are standard. They are given in Appendix A.1. Relation \Rightarrow is the transitive reflexive closure of \rightarrow .

Type System. To define the type system for CCS ℓ , we introduce typing environments. We use γ to range over sets of lock names. We write $\gamma_1 \# \gamma_2$ whenever $\gamma_1 \cap \gamma_2 = \emptyset$. We write γ, ℓ for the set $\gamma \uplus \{\ell\}$: the notation implicitly imposes $\ell \notin \gamma$.

Typing environments, written Γ , are sets of such sets, with the additional constraint that these should be pairwise disjoint. We write $\Gamma = \gamma_1, \dots, \gamma_k$, for $k \ge 1$, to mean that Γ is equal to $\{\gamma_1, \dots, \gamma_k\}$, with $\gamma_i \# \gamma_j$ whenever $i \ne j$. The γ_i s are called the *components* of Γ in this case, and dom(Γ), the domain of Γ , is defined as $\gamma_1 \cup \dots \cup \gamma_k$. We write $\Gamma_1 \# \Gamma_2$ whenever dom($\Gamma_1 \cap \Gamma_2 \cap \Gamma_3 \cap \Gamma_4 \cap \Gamma_4 \cap \Gamma_5 \cap \Gamma_5$

As for components γ , the notation Γ , γ stands for a set (of sets) that can be written as $\Gamma \uplus \{\gamma\}$. Using these two notations together, we can write Γ , γ , ℓ to refer to a typing environment containing a component that contains ℓ . We sometimes add parentheses, writing e.g. Γ , (γ, ℓ, ℓ') , to ease readability.

The typing judgement is of the form $\Gamma; \mathbb{R} \vdash P$, where \mathbb{R} is a set of lock names. If $\Gamma; \mathbb{R} \vdash P$, then \mathbb{R} is the set of locks owned by P, that must be released. Moreover any component γ of Γ intuitively corresponds to a subprocess of P that only accesses the names in γ . Here, accessing a lock name ℓ means either releasing ℓ or performing an acquire on ℓ , or both. The typing rules are as follows:

$$\frac{\text{ACQ-C}}{\Gamma, (\gamma, \ell); \mathbb{R}, \ell \vdash P} \qquad \frac{\text{REL-C}}{\{\text{flatten}(\Gamma) \uplus (\gamma, \ell)\}; \mathbb{R} \vdash \ell.P} \qquad \frac{\text{REL-C}}{\Gamma, (\gamma, \ell); \{\ell\} \vdash \overline{\ell}} \qquad \frac{\text{NEW-C}}{\Gamma, (\gamma, \ell); \mathbb{R}, \ell \vdash P} \qquad \frac{\text{PAR-C}}{\Gamma_1; \mathbb{R}_1 \vdash P_1} \qquad \frac{\Gamma_2; \mathbb{R}_2 \vdash P_2}{\Gamma_1 \bullet \Gamma_2; \mathbb{R}_1 \uplus \mathbb{R}_2 \vdash P_1 \mid P_2}$$

In rule ACQ-C, operator flatten has the effect of mergining all components in a typing environment into a single component. In particular, if $\Gamma = \{\gamma_1, \dots, \gamma_k\}$, then flatten(Γ) stands for $\gamma_1 \uplus \cdots \uplus \gamma_k$. Intuitively, the causal dependency introduced by the prefix ℓ . P induces a dependence between ℓ and all the locks in P, forcing these locks to belong to the same component.

In the typing rules, we write \mathbb{R}, ℓ for $\mathbb{R} \uplus \{\ell\}$, i.e., we suppose $\ell \notin \mathbb{R}$, otherwise the typing rule cannot be applied. Lock ℓ is added to \mathbb{R} in rule ACQ-C, to ensure that it will be released in the continuation P, and in rule NEW-C, to ensure that a newly created lock is initialised with a release. Correspondingly, rule REL-C type-checks the release of lock ℓ by imposing $\mathbb{R} = \{\ell\}$.

To type-check parallel composition, we use an operation to compose typing environments, written $\Gamma_1 \bullet \Gamma_2$. For this, we set $\emptyset \bullet \Gamma = \Gamma$ and $(\Gamma, \gamma) \bullet \Gamma' = \text{connect}(\gamma; \Gamma \bullet \Gamma')$, where $\text{connect}(\gamma; \{\gamma_1, \dots, \gamma_k\})$ is undefined as soon as there is i such that $\gamma \cap \gamma_i$ contains at least two distinct elements, and otherwise is defined as

$$\mathsf{connect}(\gamma; \{\gamma_1, \dots, \gamma_k\}) \quad = \quad \{\gamma_i : \gamma_i \# \gamma\} \ \uplus \ \{\gamma \cup \mathsf{flatten}(\{\gamma_i : \gamma_i \cap \gamma \neq \emptyset\})\}.$$

In rule PAR-C, we impose that \mathbb{R}_1 and \mathbb{R}_2 are disjoint: if lock ℓ must be released, then this is done either by P_1 or by P_2 . Together with rule REL-C, this guarantees that any $\ell \in \mathbb{R}$ is released exactly once.

We present some examples to illustrate the type system.

Example 1. Processes ℓ_1 . $(\overline{\ell_1} \mid \overline{\ell_1})$ and ℓ_1 . ℓ_2 . $\overline{\ell_1}$ cannot be typed, because both violate linearity in the usage of locks: the former releases lock ℓ_1 twice, and the latter does not release ℓ_2 after acquiring it.

Process $P_1 \stackrel{\text{def}}{=} \ell_1$. $(\overline{\ell_1} \mid \overline{\ell_2})$ acquires lock ℓ_1 , and then releases locks ℓ_1 and ℓ_2 . Let $\gamma_{12} = {\ell_1, \ell_2}$; we can derive ${\gamma_{12}}$; ${\ell_2} \vdash P_1$: locks ℓ_1 and ℓ_2 necessarily belong to the same component when typing P_1 . Similarly, we have ${\gamma_{12}}$; ${\ell_1} \vdash P_2$ with $P_2 \stackrel{\text{def}}{=} \ell_2$. $(\overline{\ell_2} \mid \overline{\ell_1})$. The typing derivations for P_1 and P_2 cannot be composed, because of the presence of γ_{12} in both, so $P_1 \mid P_2$ cannot be typed. This is appropriate, since $P_1 \mid P_2$ presents a typical deadlock situation, where ℓ_1 is needed to release ℓ_2 and conversely.

On the other hand, process $P_3 \stackrel{\text{def}}{=} \underline{\ell_1} \cdot (\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell_2 \cdot \overline{\ell_2} \mid \ell_1 \cdot \overline{\ell_1}$ can be typed: we can derive $\{\gamma_{12}\}; \{\ell_2\} \vdash \ell_1 \cdot (\overline{\ell_1} \mid \overline{\ell_2})$ and $\{\{\ell_1\}, \{\ell_2\}\}; \emptyset \vdash \ell_2 \cdot \overline{\ell_2} \mid \ell_1 \cdot \overline{\ell_1}$, and we can compose these typing derivations, yielding

 $\{\gamma_{12}\}; \{\ell_2\} \vdash P_3$. Crucially, components $\{\ell_1\}$ and $\{\ell_2\}$ are not merged in the second derivation for the composition to be possible. Using similar ideas, we can define a typable process made of three parallel components P_1, P_2, P_3 sharing a single lock, say ℓ , as long as each of the P_i uses its own locks besides ℓ .

We can derive $\{\gamma_{12}\}$; $\emptyset \vdash P_4$ with $P_4 \stackrel{\text{def}}{=} \ell_1 \cdot \ell_2 \cdot (\overline{\ell_2} \mid \overline{\ell_1})$. We observe that $P_4 \mid P_4$ cannot be typed, although $P_4 \mid P_4$ is 'no more deadlocked' than P_4 alone.

The typing rules enforce $\mathbb{R} \subseteq \operatorname{dom}(\Gamma)$ when deriving $\Gamma; \mathbb{R} \vdash P$. We say that ℓ is *available* in process P if P contains a release of ℓ which is not under an acquire on ℓ in P. Intuitively, when $\Gamma; \mathbb{R} \vdash P$ is derivable, P is a well-typed process in which all lock names in \mathbb{R} are available in P. The type system thus guarantees a linearity property on the release of names in \mathbb{R} . However, lock names are not *linear names* in the sense of [15], since there can be arbitrarily many acquire operations on a given lock. When all free lock names are available in P, i.e. $\Gamma; \operatorname{fln}(P) \vdash P$, we say that P is *complete*.

Lemma 2. The type system enjoys invariance under \equiv and subject reduction: (i) If Γ ; $\mathbb{R} \vdash P$ and $P \equiv P'$, then Γ ; $\mathbb{R} \vdash P'$. (ii) If Γ ; $\mathbb{R} \vdash P$ and $P \rightarrow P'$, then Γ ; $\mathbb{R} \vdash P'$ and $\operatorname{fln}(P') = \operatorname{fln}(P)$.

Deadlock-Freedom. Intuitively, a deadlock in CCS ℓ arises from an acquire operation that cannot be performed. We say that a *terminated process* is a parallel composition of release operations possibly under some restrictions. A process that contains at least an acquire and cannot reduce is a *stuck process*. So in particular ℓ . ℓ is stuck; the context may provide a release of ℓ , triggering the acquire on ℓ . On the other hand, if ℓ is a stuck process and complete, then ℓ is *deadlocked*: intuitively, the context cannot interact with ℓ in order to trigger an acquire operation of ℓ . Process ℓ _{dl} from Section 1 is an example of a deadlock. We show that a complete process can only reduce to a terminated process, avoiding deadlocks.

The proof of deadlock-freedom for CCS ℓ provides the structure of the proofs for deadlock-freedom in $\pi\ell$ and leak-freedom in $\pi\ell$ w. It relies on progress: any typable process can reduce to reach a terminated process. We first present some lemmas related to the absence of cyclic structures in CCS ℓ .

Lemma 3 (Lock-connected processes). We say that P is lock-connected if Γ ; $\mathbb{R} \vdash P$ implies $\Gamma = \Gamma'$, γ for some Γ' , γ , with $\operatorname{fln}(P) \subseteq \gamma$. In this situation, we also have $\{\gamma\}$; $\mathbb{R} \vdash P$. If P and Q are lock-connected and $\operatorname{fln}(P) \cap \operatorname{fln}(Q)$ contains at least two distinct names, then $P \mid Q$ cannot be typed.

The property in Lemma 3 does not hold if P and Q are not lock-connected: take for instance $P = Q = \ell_1.\overline{\ell_1} \mid \ell_2.\overline{\ell_2}$, then we can derive $\{\{\ell_1\},\{\ell_2\}\};\emptyset \vdash P \mid Q$. By the typing rule ACQ-C, any process of the form $\ell.P$ is lock-connected. A typical example of a lock-connected process is $\ell_1.(\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell_2.(\overline{\ell_2} \mid \overline{\ell_3})$: here $\gamma = \{\ell_1,\ell_2,\ell_3\}$. Processes similar to this one are used in the following lemma.

Lemma 4 (No cycle). We write $P \stackrel{\ell}{\longleftrightarrow} Q$ when $\ell \in \text{fln}(P) \cap \text{fln}(Q)$. Suppose there are k > 1 pairwise distinct names ℓ_1, \ldots, ℓ_k , and processes P_1, \ldots, P_k such that $\ell_i.P_i \stackrel{\ell_i}{\longleftrightarrow} \ell_{(i+1) \mod k}.P_{(i+1) \mod k}$ for $1 \le i \le k$. Then $P_1 \mid \ldots \mid P_k$ is not typable.

We use notation $\prod_i P_i$ for the parallel composition of processes P_i .

Lemma 5 (Progress). *If* Γ ; fln(P) $\vdash P$, then either $P \to P'$ for some P', or $P \equiv (v\tilde{\ell}) \prod_i \overline{\ell_i}$ where the ℓ_i s are pairwise distinct.

Proof. Write $P \equiv (v\tilde{\ell})P_0$ with $P_0 = \prod_i \overline{\ell_i} \mid \prod_j \ell_j.P_j$. We let Q_j stand for $\ell_j.P_j$, and suppose that there is at least one Q_j . We show that under this hypothesis P_0 can reduce.

If $\ell_i = \ell_j$ for some i, j, then P_0 can reduce. We suppose in the following that this is not the case, and consider one of the Q_j s. By typing, there exists a unique occurrence of ℓ_j available in P_0 . By hypothesis, this occurrence is not among the $\overline{\ell_i}$ s. Therefore, ℓ_j is available in $Q_{j'}$ for some unique j' with $j \neq j'$.

We construct a graph having one vertex for each of the Q_j s. We draw an edge between Q_j and $Q_{j'}$ when ℓ_j is available in $Q_{j'}$. By the reasoning we just made, each vertex is related to at least one other vertex. So the graph necessarily contains a cycle. We can apply Lemma 4 to derive a contradiction.

We make two remarks about the construction of the graph. First, two Q_j s may start with an acquire at the same name. The corresponding vertices will have edges leading to the same $Q_{j'}$, and the construction still works. Second, if there is only one Q_j , then the available release of ℓ_j can synchronise with Q_j . \square

By Lemma 5, we have that any typable process is not deadlocked. Thus, by subject reduction, we can prove deadlock-freedom.

Proposition 6 (Deadlock-freedom). *If* Γ ; $\mathbb{R} \vdash P$ *and* $P \Rightarrow P'$, *then* P' *is not deadlocked.*

Remark 7. As CCS ℓ is finite, deadlock-freedom ensures that no acquire operation waits forever in a complete typable process, and every complete process reduces to a terminated process: if Γ ; fln(P) $\vdash P$, then $P \Rightarrow (\nu \ell) \prod_i \ell_i$ where the ℓ_i s are pairwise distinct.

2.2 $\pi \ell$: Deadlock-Freedom for Higher-Order Locks

Syntax and Operational Semantics of $\pi\ell$. $\pi\ell$ extends CCS ℓ with the possibility to store *values*, which can be either booleans or locks, in locks. In this sense, $\pi\ell$ features higher-order locks. Processes in $\pi\ell$ are defined as follows:

$$P \quad ::= \quad \ell(\ell').P \mid \overline{\ell}\langle v \rangle \mid (v\ell)P \mid P_1 \mid P_2 \mid \mathbf{0} \mid [v = v']P_1, P_2.$$

v,v' denote *values*, defined by $v := \ell \mid b$, where $b := tt \mid ff$ is a boolean value. In addition to $\ell,\ell'...$, we sometimes use also x,y... to range over lock names, to suggest a specific usage, like, e.g. in $\ell(x).P$.

Process $\overline{\ell}\langle\ell'\rangle$ is a *release of* ℓ , and $\ell(\ell').P$ is an *acquire on* ℓ ; we say in both cases that ℓ is the *subject* (or that ℓ occurs in subject position) and ℓ' is the *object*. Restriction and the acquire prefix act as binders, giving rise to the notion of bound and free names. As in $CCS\ell$, we write fln(P) for the set of free lock names of P. $P\{\nu/\ell\}$ is the process obtained by replacing every free occurrence of ℓ with ν in P. We say that an occurrence of a process Q in P is *guarded* if the occurrence is under an acquire prefix, otherwise it is said *at top-level* in P. Additional operators w.r.t. $CCS\ell$ are the inactive process, $\mathbf{0}$, and value comparison: $[\nu = \nu']P_1, P_2$ behaves like P_1 if values ν and ν' are equal, and like P_2 otherwise.

Structural congruence in $\pi \ell$ is defined by adding the following axioms to \equiv in CCS ℓ :

$$P \mid \mathbf{0} \equiv P$$
 $(v\ell)\mathbf{0} \equiv \mathbf{0}$ $[v=v]P_1, P_2 \equiv P_1$ $[v=v']P_1, P_2 \equiv P_2 \text{ if } v \neq v'$

The last axiom above cannot be used under an acquire prefix: see Appendix A.3 for the definition of \equiv . *Execution contexts*, are defined by $E := [\cdot] \mid E \mid P \mid (\nu \ell)E$. The axiom for reduction in $\pi \ell$ is:

$$\overline{\overline{\ell}\langle v\rangle \mid \ell(\ell').P \rightarrow P\{v/\ell'\}}$$

 \Rightarrow is the reflexive transitive closure of \rightarrow . Labelled transitions, written $P \xrightarrow{\mu} P'$, use actions μ defined by $\mu ::= \ell(\nu) \mid \overline{\ell}\langle \nu \rangle \mid \overline{\ell}(\ell') \mid \tau$, and are standard [26]—we recall the definition in Appendix A.3.

Figure 1: Typing rules for $\pi \ell$

The type system. We enforce a sorting discipline for names [17], given by $V := \text{bool} \mid L$ and $\Sigma(L) = V$: values, that are stored in locks, are either booleans or locks. We consider that all processes we write obey this discipline, which is left implicit. This means for instance that when writing $\overline{\ell}\langle v \rangle$, ℓ and v have appropriate sorts; and similarly for $\ell(\ell').P$. In $[v = v']P_1, P_2$, we only compare values with the same sort.

The typing judgement is written Γ ; $\mathbb{R} \vdash P$, where Γ and \mathbb{R} are defined like for CCS ℓ . We adopt the convention that if ν is a boolean value, then γ, ν is just γ , and similarly, γ, ℓ is just γ if the sort of ℓ is bool. The operation $\Gamma_1 \bullet \Gamma_2$ is the same as for CCS ℓ .

The typing rules for $\pi\ell$ are presented in Figure 1. Again, in rules ACQ and NEW, writing \mathbb{R},ℓ imposes $\ell\notin\mathbb{R}$, otherwise the rule cannot be applied. Similarly, the notation γ,ℓ,ℓ' is only defined when $\gamma\#\{\ell,\ell'\}$ and $\ell\neq\ell'$. Rule REL describes the release of a lock containing either a lock or a boolean value: in the latter case, using the convention above, the conclusion of the rule is $\{\{\ell\}\}; \{\ell\} \vdash \overline{\ell}\langle b\rangle$. In rules ACQ and REL, the subject and the object of the operation should belong to the same component. In CCS ℓ , only prefixing yields such a constraint.

In rule MAT, we do not impose $\{v,v'\} \in \text{dom}(\Gamma)$. A typical example of a process that uses name comparison is $[\ell_1 = \ell_2] \overline{\ell} \langle \text{tt} \rangle, \overline{\ell} \langle \text{ff} \rangle$: in this process, ℓ_1 and ℓ_2 intuitively represent no threat of a deadlock.

Before presenting the properties of the type system, we make some comments on the discipline it imposes on π -calculus names when they are used as locks.

Remark 8 (An acquired lock cannot be stored). In $\pi\ell$, the obligation to release a lock cannot be transmitted. Accordingly, $\ell' \notin \mathbb{R} = \{\ell\}$ in rule REL, and a process like $\ell(\ell')$. $\overline{\ell_1}\langle\ell\rangle$ cannot be typed. We return to this point after Proposition 11.

Remark 9 (Typability of higher-order locks). Locks are a particular kind of names of the asynchronous π -calculus ($A\pi$). Acquiring a lock that has been stored in another lock boils down to performing a communication in $A\pi$. We discuss how such communications can occur between typed processes.

In rule REL, ℓ and ℓ' must belong to the same component of Γ . So intuitively, if a process contains $\overline{\ell}\langle\ell'\rangle$, this release is the only place where these locks are used 'together'. A reduction involving a well-typed process containing this release therefore looks like

$$(\overline{\ell}\langle \ell' \rangle \mid P) \quad | \quad (\ell(x).Q \mid Q') \quad \rightarrow \quad P \mid Q\{\ell'/x\} \mid Q'.$$

Parentheses are used to suggest an interaction between two processes; $\overline{\ell}\langle\ell'\rangle \mid P$ performs the release, and $\ell(x)$. $Q \mid Q'$ performs the acquire. Process P, which intuitively is the continuation of the release, may use locks ℓ and ℓ' , but not together, and similarly for Q'. For instance we may have $P = P_{\ell} \mid P_{\ell'}$, where ℓ' does not occur in P_{ℓ} , and vice-versa for $P_{\ell'}$. Note also that ℓ' is necessarily fresh for $\ell(x)$. Q': otherwise,

typability of $\ell(x)$. Q' would impose ℓ and ℓ' to be in the same component, which would forbid the parallel composition with $\overline{\ell}\langle\ell'\rangle$.

Depending on how P,Q and Q' are written, we can envisage several patterns of usages of locks ℓ and ℓ' . A first example is ownership transfer (or delegation): $\ell' \notin \operatorname{fln}(P)$, that is, P renounces usage of ℓ' . ℓ' can be used in Q. Note that typing actually also allows $\ell' \in \operatorname{fln}(Q')$, i.e., the recipient already knows ℓ' .

A second possibility could be that ℓ is used linearly, in the sense that there is exactly one acquire on ℓ . In this case, we necessarily have $\ell \notin \operatorname{fln}(P) \cup \operatorname{fln}(Q')$ —note that a release of ℓ is available in Q, by typing. Linearity of ℓ means here that exactly one interaction takes place at ℓ . After that interaction, the release on ℓ contained in Q is inert, in the sense that no acquire can synchronise with it. We believe that this form of linearity can be used to encode binary session types in an extended version of $\pi\ell$, including variants and polyadicity, along the lines of [14, 3, 4].

The type system for $\pi\ell$ satisfies the same properties as in CCS ℓ (Lemma 2): invariance under structural congruence, merging components and subject reduction. We also have progress and deadlock-freedom:

Lemma 10 (Progress). *Suppose* Γ ; fln(P) $\vdash P$, and P is not structurally equivalent to **0**. Then

- either there exists P' such that $P \rightarrow P'$,
- or $P \equiv (v\widetilde{\ell})(\Pi_i\overline{\ell_i}v_i)$ where the ℓ_i s are pairwise distinct.

Like in CCS ℓ , a deadlocked process in $\pi\ell$ is defined as a complete process that is stuck.

Proposition 11 (Deadlock-freedom). *If* Γ ; $\mathbb{R} \vdash P$ *and* $P \Rightarrow P'$, *then* P' *is not deadlocked.*

The proof of deadlock-freedom is basically the same as for CCS ℓ . The reason for that is that although the object part of releases plays a role in the typing rules, it is not relevant to establish progress (Lemma 10). This is the case because in $\pi\ell$, it is not possible to store an acquired lock in another lock (Remark 8).

It seems difficult to extend the type system in order to allow processes that transmit the release obligation on a lock. This would make it possible to type-check, e.g., process $\ell(\ell')$. $\overline{\ell_1}\langle\ell\rangle$, that does not release lock ℓ but instead stores it in ℓ_1 . Symmetrically, a process accessing ℓ at ℓ_1 would be in charge of releasing both ℓ_1 and ℓ . In such a framework, a process like $(v\ell_1)(\overline{\ell_1}\langle\ell\rangle \mid \ell(x).\overline{\ell}\langle x\rangle)$ would be deadlocked, because the inert release $\overline{\ell_1}\langle\ell\rangle$ contains the release obligation on ℓ_1 . The type system in Section 3 makes it possible to transmit the obligation to perform a release (and similarly for a wait).

Remark 12. Similarly to Remark 7, we have that Γ ; $\operatorname{fln}(P) \vdash P$ implies $P \Rightarrow (v\widetilde{\ell})(\Pi_i\overline{\ell_i}v_i)$ where the ℓ_i s are pairwise distinct. As a consequence, the following holds: if Γ ; $\operatorname{fln}(P) \vdash P$, then for any $\ell \in \operatorname{fln}(P)$, $P \Rightarrow \stackrel{\mu}{\rightarrow}$, where μ is a release of ℓ . This statement would be better suited if infinite computations were possible in $\pi\ell$. We leave the investigation of such an extension of $\pi\ell$ for future work.

2.3 Behavioural Equivalence in $\pi \ell$

We introduce typed barbed equivalence (\simeq) and typed bisimilarity (\approx) for $\pi\ell$. We show that \approx is a sound technique to establish \simeq , and present several examples of (in)equivalences between $\pi\ell$ processes.

2.3.1 Barbed Equivalence and Labelled Semantics for $\pi \ell$

A typed relation in $\pi \ell$ is a set of quadruples of the form $(\Gamma, \mathbb{R}, P, Q)$ such that $\Gamma; \mathbb{R} \vdash P$ and $\Gamma; \mathbb{R} \vdash Q$. When a typed relation \mathscr{R} contains $(\Gamma, \mathbb{R}, P, Q)$, we write $\Gamma; \mathbb{R} \vdash P \mathscr{R} Q$. We say that a typed relation \mathscr{R} is symmetric if $\Gamma; \mathbb{R} \vdash P \mathscr{R} Q$ implies $\Gamma; \mathbb{R} \vdash Q \mathscr{R} P$.

Deadlock-freedom has two consequences regarding the definition of barbed equivalence in $\pi\ell$, noted \simeq . First, only complete processes should be observed, because intuitively a computation in $\pi\ell$ should not be blocked by an acquire operation that cannot be executed.

Second, Proposition 11 entails that all weak barbs in the sense of $A\pi$ can always be observed in $\pi\ell$. In $A\pi$, a weak barb at n corresponds to the possibility to reduce to a process in which an output at channel n occurs at top-level. We need a stronger notion of barb, otherwise \simeq would be trivial. That behavioural equivalence in $\pi\ell$ is not trivial is shown for instance by the presence of non-determinism. Consider indeed process $P_c \stackrel{\text{def}}{=} (v\ell) \Big(\ell(x). (\overline{c}\langle x \rangle \mid \overline{\ell}\langle x \rangle) \mid \ell(y). \overline{\ell}\langle \text{ff} \rangle \mid \overline{\ell}\langle \text{tt} \rangle \Big)$. Then $P_c \Rightarrow \overline{c}\langle \text{tt} \rangle$ and $P_c \Rightarrow \overline{c}\langle \text{ff} \rangle$ (up to the cancellation of an inert process of the form $(v\ell)\overline{\ell}\langle \text{b}\rangle$). We therefore include the object part of releases in barbs. We write $P \downarrow_{\overline{\ell}\langle \ell' \rangle}$ if $P \stackrel{\overline{\ell}\langle \ell' \rangle}{\longrightarrow}$, and $P \downarrow_{\overline{\ell}(v)}$ if $P \stackrel{\overline{\ell}\langle \ell' \rangle}{\longrightarrow}$. We use η to range over barbs, writing $P \downarrow_{\eta}$; the weak version of the predicate, defined as $\Rightarrow \downarrow_{\eta}$, is written $P \downarrow_{\eta}$.

Definition 13 (Barbed equivalence in $\pi \ell$, \simeq). A symmetric typed relation \mathscr{R} is a typed barbed bisimulation if Γ ; $\mathbb{R} \vdash P\mathscr{R}Q$ implies the three following properties:

- 1. whenever P,Q are complete and $P \to P'$, there is Q' s.t. $Q \Rightarrow Q'$ and $\Gamma; \mathbb{R} \vdash P' \mathcal{R} Q'$;
- 2. for any η , if P,Q are complete and $P\downarrow_{\eta}$ then $Q\downarrow_{\eta}$;
- 3. for any E, Γ', \mathbb{R}' s.t. $\Gamma'; \mathbb{R}' \vdash E[P]$ and $\Gamma'; \mathbb{R}' \vdash E[Q]$, and E[P], E[Q] are complete, we have $\Gamma'; \mathbb{R}' \vdash E[P] \mathscr{R} E[Q]$.

Typed barbed equivalence, written \simeq , is the greatest typed barbed bisimulation.

Lemma 14 (Observing only booleans). We use o, o', ... for lock names that are used to store boolean values. We define \simeq_o as the equivalence defined as in Definition 13, but restricting the second clause to barbs of the form $\downarrow_{\overline{o}(b)}$ and $\downarrow_{\overline{o}(b)}$. Relation \simeq_o coincides with \simeq .

To define typed bisimilarity, we introduce *type-allowed transitions*. The terminology means that we select among the untyped transitions those that are fireable given the constraints imposed by types.

Definition 15 (Type-allowed transitions). When $\Gamma; \mathbb{R} \vdash P$, we write $[\Gamma; \mathbb{R}; P] \xrightarrow{\mu} [\Gamma'; \mathbb{R}'; P']$ if $P \xrightarrow{\mu} P'$ and one of the following holds:

- 1. $\mu = \tau$, in which case $\mathbb{R}' = \mathbb{R}$ and $\Gamma' = \Gamma$;
- 2. $\mu = \overline{\ell}\langle v \rangle$, in which case $(\gamma, \ell, v) \in \Gamma$ for some γ , and $\mathbb{R}', \ell = \mathbb{R}$, $\Gamma' = \Gamma$;
- 3. $\mu = \overline{\ell}(\ell')$, in which case $\Gamma = \Gamma_0, (\gamma, \ell)$ for some $\Gamma_0, \gamma, \Gamma' = \Gamma_0, (\gamma, \ell, \ell')$, and we have $\mathbb{R}', \ell = \mathbb{R}, \ell'$;
- 4. $\mu = \ell(\nu)$, in which case there are Γ_0, \mathbb{R}_0 s.t. $\Gamma_0; \mathbb{R}_0 \vdash P \mid \overline{\ell}\langle \nu \rangle$, and $\Gamma' = \Gamma_0, \mathbb{R}' = \mathbb{R}_0$.

In item 3, ℓ is removed from the $\mathbb R$ component, and ℓ' is added: it is P''s duty to perform the release of ℓ' , the obligation is not transmitted. An acquire transition involving a higher-order lock merges two distinct components in the typing environment: if $[\Gamma_0, (\gamma, \ell), (\gamma', \ell'); \mathbb{R}; P] \xrightarrow{\ell(\ell')} [\Gamma'; \mathbb{R}'; P']$ (item 4 above), then $\Gamma' = \Gamma_0, (\gamma \uplus \gamma' \uplus \{\ell, \ell'\})$ and $\mathbb{R}' = \mathbb{R}, \ell$ (and in particular $\ell \notin \mathbb{R}$).

Lemma 16 (Subject Reduction for type-allowed transitions). *If* $[\Gamma; \mathbb{R}; P] \xrightarrow{\mu} [\Gamma'; \mathbb{R}'; P']$, *then* $\Gamma'; \mathbb{R}' \vdash P'$.

Definition 17 (Typed bisimilarity, \approx). A typed relation \mathcal{R} is a typed bisimulation if Γ ; $\mathbb{R} \vdash P\mathcal{R}Q$ implies that whenever $[\Gamma; \mathbb{R}; P] \xrightarrow{\mu} [\Gamma'; \mathbb{R}'; P']$, we have

1. either
$$Q \stackrel{\hat{\mu}}{\Rightarrow} Q'$$
 and Γ' ; $\mathbb{R}' \vdash P' \mathcal{R} Q'$ for some Q'

D. Hirschkoff and E. Prebet

85

2. or μ is an acquire $\ell(v)$, $Q \mid \overline{\ell}\langle v \rangle \Rightarrow Q'$ and Γ' ; $\mathbb{R}' \vdash P' \mathcal{R} Q'$ for some Q',

and symmetrically for the type-allowed transitions of Q.

Typed bisimilarity, written \approx , is the largest typed bisimulation.

We write Γ ; $\mathscr{R} \vdash P \approx Q$ when $(\Gamma; \mathbb{R}, P, Q) \in \approx$. If Γ ; $\mathbb{R} \vdash P \approx Q$ does not hold, we write Γ ; $\mathbb{R} \vdash P \not\approx Q$, and similarly for Γ ; $\mathbb{R} \vdash P \not\simeq Q$.

Proposition 18 below states that relation \approx provides a sound proof technique for \simeq . The main property to establish this result is that \approx is preserved by parallel composition: Γ_0 ; $\mathbb{R}_0 \vdash P \approx Q$ implies that for all T, whenever Γ ; $\mathbb{R} \vdash P \mid T$ and Γ ; $\mathbb{R} \vdash Q \mid T$, we have Γ ; $\mathbb{R} \vdash P \mid T \approx Q \mid T$.

Proposition 18 (Soundness). *For any* Γ , \mathbb{R} , P, Q, *if* Γ ; $\mathbb{R} \vdash P \approx Q$, *then* Γ ; $\mathbb{R} \vdash P \simeq Q$.

The main advantage in using \approx to establish equivalences for \simeq is that we can reason directly on processes, even if they are not complete.

2.3.2 Examples of Behavioural Equivalence in $\pi \ell$

Example 19. We discuss some equivalences for \simeq .

The equivalence $\{\{\ell\}\}; \emptyset \vdash \ell(x).\overline{\ell}\langle x\rangle \simeq \mathbf{0}$, which is typical of $A\pi$, holds in $\pi\ell$. This follows directly from the definition of typed bisimilarity, and soundness (Proposition 18).

We now let $P \stackrel{\text{def}}{=} \ell(x)$. $(\overline{\ell_0}\langle \mathsf{tt} \rangle \mid \overline{\ell}\langle x \rangle)$ and $Q \stackrel{\text{def}}{=} \overline{\ell_0}\langle \mathsf{tt} \rangle$, and consider whether we can detect the presence of a 'forwarder' at ℓ when its behaviour is interleaved with another process. P and Q have different barbs—they are obviously not complete. It turns out that $\{\{\ell,\ell_0\}\};\{\ell_0\} \vdash \ell(x). (\overline{\ell_0}\langle \mathsf{tt} \rangle \mid \overline{\ell}\langle x \rangle) \not\simeq \overline{\ell_0}\langle \mathsf{tt} \rangle$. Indeed, let us consider the context

$$E \stackrel{\text{def}}{=} [\cdot] \mid \ell_0(y).w(\underline{\ }).(\overline{w}\langle \mathsf{tt} \rangle \mid \overline{\ell_0}\langle y \rangle) \mid w'(\underline{\ }).(\overline{w'}\langle \mathsf{tt} \rangle \mid \overline{\ell}\langle v \rangle) \mid \overline{w}\langle \mathsf{ff} \rangle \mid \overline{w'}\langle \mathsf{ff} \rangle,$$

where w, w' are fresh names and v is a value of the appropriate sort. We have $E[Q] \Rightarrow Q'$ with $Q' \not\downarrow_{\overline{w}(ff)}$ and $Q' \downarrow_{\overline{w}'(ff)}$. On the other hand, for any P' s.t. $E[P] \Rightarrow P'$, if $P' \not\downarrow_{\overline{w}(ff)}$, then $P' \not\downarrow_{\overline{w}'(ff)}$.

Contexts like E above make it possible to detect when the process in the hole has some interaction (here, with locks ℓ and ℓ_0).

Using similar ideas, we can prove that

$$\Gamma; \mathbb{R} \vdash \ell_1(x).\ell_2(y).P \not\simeq \ell_2(y).\ell_1(x).P$$
 for appropriate Γ and \mathbb{R} .

Indeed, let us define $E_w \stackrel{\text{def}}{=} \overline{w}\langle \mathsf{ff} \rangle \mid w(_).([\cdot] \mid \overline{w}\langle \mathsf{tt} \rangle)$, where $_$ stands for an arbitrary lock name, that is not used. We can use the context $[\cdot] \mid E_{w_2}[\overline{\ell_2}\langle v_2 \rangle] \mid \overline{\ell_1}\langle v_1 \rangle \mid \ell_1(z).E_{w_1}[\overline{\ell_1}\langle z \rangle]$, for fresh names w_1, w_2 and appropriate values v_1, v_2 , to detect the order in which acquires on ℓ_1 and ℓ_2 are made.

In the next two examples, we show equivalences that hold because we work in a typed setting.

Example 20. Suppose Γ ; \mathbb{R} , $\ell \vdash \ell(x)$. $P \mid \ell'(y)$. $(\overline{\ell}\langle v \rangle \mid Q)$. Then we have

$$\Gamma; \mathbb{R}, \ell \vdash \ell(x).P \mid \ell'(y).(\overline{\ell}\langle v \rangle \mid Q) \approx \ell'(y).(\overline{\ell}\langle v \rangle \mid Q \mid \ell(x).P),$$

because intuitively the acquire on ℓ cannot be triggered by the context, due to the presence of a release at ℓ in the process. (We remark in passing that $\Gamma; \mathbb{R}, \ell \vdash \ell(x).P \mid \ell'(y).(\overline{\ell}\langle v \rangle \mid Q)$ iff $\Gamma; \mathbb{R}, \ell \vdash \ell'(y).(\overline{\ell}\langle v \rangle \mid Q \mid \ell(x).P)$, and in this case Γ contains a component of the form $(\gamma, \ell, \ell', v).$)

This law can be generalised as follows. We say that ℓ is available in a context C if the hole does not occur in C neither under a binder for ℓ , nor under an acquire on ℓ . So for instance ℓ is not available in $(v\ell)[\cdot]$, in $\ell_0(\ell)$. $[\cdot]$ or in $\ell(x)$. $[\cdot]$, and ℓ is available in $\ell(x)$. $[\ell(x)]$ $[\ell($

$$\Gamma; \mathbb{R} \vdash \ell(x).P \mid C[\overline{\ell}\langle v \rangle] \approx C[\overline{\ell}\langle v \rangle \mid \ell(x).P]$$
 for appropriate Γ and \mathbb{R} .

Example 21. Consider the following processes:

$$\begin{array}{lll} P_1 &=& (\nu\ell_1)\big(\,\ell_1.\,\ell_2.\,(\overline{\ell_1}\,\,|\,\,\overline{\ell_2})\,\,|\,\,\,\ell(x).\,\ell_1.\,x.\,(\overline{\ell_1}\,\,|\,\,\overline{x}\,\,|\,\,\overline{\ell}\langle x\rangle)\,\,|\,\,\,\overline{\ell_1}\,\,|\,\,\overline{\ell_2}\,\big) \\ P_2 &=& (\nu\ell_1)\big(\,\ell_1.\,\ell_2.\,(\overline{\ell_1}\,\,|\,\,\overline{\ell_2})\,\,|\,\,\,\ell(x).\,x.\,\ell_1.\,(\overline{\ell_1}\,\,|\,\,\overline{x}\,\,|\,\,\overline{\ell}\langle x\rangle)\,\,|\,\,\,\overline{\ell_1}\,\,|\,\,\overline{\ell_2}\,\big) \end{array}$$

Here we use a CCS-like syntax, to ease readability. This notation means that acquire operations are used as forwarders, i.e., the first component of P_1 and P_2 should be read as $\ell_1(y_1).\ell_2(y_2).(\overline{\ell_1}\langle y_1\rangle \mid \overline{\ell_2}\langle y_2\rangle)$. Moreover, the two releases available at top-level are $\overline{\ell_1}\langle \mathsf{tt}\rangle \mid \overline{\ell_2}\langle \mathsf{tt}\rangle$, and similarly for $\overline{x}\langle \mathsf{tt}\rangle$ (the reasoning also holds if ℓ_1 and ℓ_2 are higher-order locks).

In the pure π -calculus, P_1 and P_2 are not equivalent, because ℓ_2 can instantiate x in the acquire on ℓ . We can show $\{\{\ell_2,\ell\}\}; \{\ell_2\} \vdash P_1 \approx P_2$ in $\pi\ell$, because the transition $\xrightarrow{\ell(\ell_2)}$ is ruled out by the type system.

3 $\pi \ell w$, a Leak-Free Asynchronous π -Calculus

3.1 Adding Lock Deallocation

 $\pi \ell$ w is obtained from $\pi \ell$ by adding the *wait construct* $\ell((\ell'))$. P to the grammar of $\pi \ell$. As announced in Section 1, the following reduction rule describes how wait interacts with a release:

$$\frac{1}{(\nu\ell)(\overline{\ell}\langle\nu\rangle\ |\ \ell((\ell')).P)\ \rightarrow\ P\{\nu/\ell'\}}\ \ell\notin \mathrm{fln}(P)$$

The wait instruction deallocates the lock. The continuation may use ℓ' , the final value of the lock. We say that $\ell((\ell'))$ is a *wait on* ℓ , and ℓ' is bound in $\ell((\ell'))$. P.

Types in $\pi \ell w$, written T, T', \ldots , are defined by $T ::= bool | \langle T \rangle_{rw}$, and typing hypotheses are written $\ell : T$. In $\ell : \langle T \rangle_{rw}$, rw is called the *usage* of ℓ , and $r, w \in \{0,1\}$ are the release and wait *obligations*, respectively, on lock ℓ . So for instance a typing hypothesis of the form $\ell : \langle T \rangle_{10}$ means that ℓ must be used to perform a release and cannot be used to perform a wait. An hypothesis $\ell : \langle T \rangle_{00}$ means that ℓ can only be used to perform acquire operations. This structure for types makes it possible to transmit the wait and release obligations on a given lock name via higher-order locks.

Our type system ensures that locks are properly deallocated. In contrast to $\pi \ell$, this allows acquired locks to be stored without creating deadlocks. For example, a process like $(\nu \ell_1)(\overline{\ell_1}\langle \ell \rangle \mid \ell(x).\overline{\ell}\langle x \rangle)$ is deadlocked if ℓ_1 stores the release obligation of ℓ ; however, it cannot be typed as it lacks the wait on ℓ_1 . Adding a wait, e.g. $\ell_1((\ell)).\overline{\ell}\langle \nu \rangle$ removes the deadlock.

Typing environments have the same structure as in Section 2, except that components γ are sets of typing hypotheses instead of simply sets of lock names. dom(Γ) is defined as the set of lock names for which Γ contains a typing hypothesis. We write $\Gamma(\ell) = T$ if the typing hypothesis ℓ : T occurs in Γ .

We reuse the notation for composition of typing environments. $\Gamma_1 \bullet \Gamma_2$ is defined like in Section 2.1, using the connect operator, to avoid cyclic structures in the sharing of lock names. Additionally, when merging components, we compose typing hypotheses. For any ℓ , if $\ell : \langle \mathsf{T}_1 \rangle_{r_1w_1} \in \mathsf{dom}(\Gamma_1)$ and $\ell : \langle \mathsf{T}_2 \rangle_{r_2w_2} \in \mathsf{dom}(\Gamma_2)$, the typing hypothesis for ℓ in $\Gamma_1 \bullet \Gamma_2$ is $\ell : \langle \mathsf{T} \rangle_{(r_1+r_2)(w_1+w_2)}$, and is defined only if $\mathsf{T} = \mathsf{T}_1 = \mathsf{T}_2$, $r_1 + r_2 \leq 1$ and $w_1 + w_2 \leq 1$.

The typing rules are given in Figure 2. The rules build on the rules for $\pi\ell$, and rely on usages to control the release and wait obligations. In particular, the set \mathbb{R} in Figure 1 corresponds to the set of locks whose usage is of the form 1w in this system. To type-check an acquire, we can have usage 00, but also 01, as in, e.g., $\ell(\ell')$. $(\overline{\ell}\langle\ell'\rangle \mid \ell((x)).P)$. In rule REL-W, we impose that all typing hypotheses in Γ_{00} (resp. γ_{00}) have the form ℓ : $\langle T \rangle_{00}$.

$$\frac{\text{ACQ-W}}{\Gamma, (\gamma, \ell : \langle \mathsf{T} \rangle_{1w}, \ell' : \mathsf{T}) \vdash P} \\ \frac{\Gamma, (\gamma, \ell : \langle \mathsf{T} \rangle_{1w}, \ell' : \mathsf{T}) \vdash P}{\{\text{flatten}(\Gamma) \uplus (\gamma, \ell : \langle \mathsf{T} \rangle_{0w})\} \vdash \ell(\ell').P} \\ \frac{\text{ReL-W}}{\Gamma_{00}, (\gamma_{00}, \ell : \langle \mathsf{T} \rangle_{10}, \nu : \mathsf{T}) \vdash \overline{\ell} \langle \nu \rangle} \\ \frac{\text{WAIT-W}}{\{\gamma, \ell' : \mathsf{T}\} \vdash P} \\ \frac{\{\gamma, \ell' : \mathsf{T}\} \vdash P}{\{\gamma, \ell : \langle \mathsf{T} \rangle_{01}\} \vdash \ell((\ell')).P} \\ \frac{PAR-W}{\Gamma, (\gamma, \ell : \langle \mathsf{T} \rangle_{11}) \vdash P} \\ \frac{\Gamma, (\gamma, \ell : \langle \mathsf{T} \rangle_{11}) \vdash P}{\Gamma, \gamma \vdash (\nu \ell) P} \\ \frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \bullet \Gamma_2 \vdash P_1 \mid P_2} \\ \frac{MAT-W}{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2} \\ \frac{\Gamma}{\Gamma \vdash [\nu = \nu'] P_1, P_2} \\ \frac{PAR-W}{\Gamma \vdash P_1 \qquad \Gamma_2 \vdash P_2} \\ \frac{PAR-W}{\Gamma \vdash P$$

Figure 2: Typing rules for $\pi \ell w$

Several notions introduced for the type system of Section 2 have to be adapted in the setting of $\pi \ell w$. While in Section 2 we simply say that a lock ℓ is available, here we distinguish whether a release of ℓ or a wait on ℓ is available. If P has a subterm of the form $\ell((x))$. Q that does not occur under a binder for ℓ , we say that a wait on ℓ is available in P. If $\overline{\ell}\langle v\rangle$ occurs in some process P and this occurrence is neither under a binder for ℓ nor under an acquire on ℓ , we say that a release of ℓ is available in P. In addition, a release of ℓ (resp. wait on ℓ) is available in P also if P contains a release of the form $\overline{\ell_0}\langle \ell \rangle$, which does not occur under a binder for ℓ , and if ℓ 's type is of the form $\langle \mathsf{T} \rangle_{1w}$ (resp. $\langle \mathsf{T} \rangle_{r1}$).

Like in $\pi\ell$, a deadlocked process in $\pi\ell$ w is a complete process that is stuck. The notion of complete process has to be adapted in order to take into account the specificities of $\pi\ell$ w. First, the process should not be stuck just because a restriction is missing in order to trigger a name deallocation. Second, we must consider the fact that release and wait obligations can be stored in locks in $\pi\ell$ w. As a consequence, when defining complete processes in $\pi\ell$ w, we impose some constraints on the free lock names of processes.

In $\pi \ell$ w, we say that Γ is complete if for any $\ell \in \text{dom}(\Gamma)$, either $\Gamma(\ell) = \langle \text{bool} \rangle_{10}$ or $\Gamma(\ell) = \langle \langle \mathsf{T} \rangle_{00} \rangle_{10}$ for some T . To understand this definition, suppose $\Gamma \vdash P$ with Γ complete. Then we have, for any free lock name ℓ of P: (i) the release of ℓ is available in P; (ii) this release does not carry any obligation; (iii) the wait on ℓ is *not* available in P. The latter constraint means that if a P contains a wait on some lock, then this lock should be restricted.

The notion of leak-freedom we use is inspired from [12]. In our setting, a situation where some lock ℓ is released and will never be acquired again can be seen as a form of memory leak. We say that P leaks ℓ if $P \equiv (v\ell)(P' \mid \overline{\ell}\langle v\rangle)$ with $\ell \notin \text{fln}(P')$. P has a leak if P leaks ℓ for some ℓ , and is leak-free otherwise.

Lemma 22 (Progress). *If* $\Gamma \vdash P$ *and* Γ *is complete, then either* $P \to P'$ *for some* P', or $P \equiv (v\widetilde{\ell})(\Pi_i \overline{\ell_i} v_i)$ where the ℓ_i s are pairwise distinct.

For lack of space, the proof is presented in Appendix B. Again, it follows the lines of the proof of Lemma 5. To construct a graph containing necessarily a cycle, we associate to every acquire of the form $\ell(x)$. Q an available release of ℓ , which might occur in a release of the form $\overline{\ell'}\langle\ell\rangle$, if ℓ' carries the release obligation. Similarly, to every wait $\ell((x))$. Q, we associate an available release, or, if a release $\overline{\ell}\langle\nu\rangle$ occurs at top-level, an acquire on ℓ , that necessarily exists otherwise a reduction could be fired. Finally, using a similar reasoning, to every release of ℓ at top-level, we associate a wait on ℓ , or an acquire on ℓ .

A consequence of Lemma 22 is that $P \Rightarrow \mathbf{0}$ when $\emptyset \vdash P$.

Proposition 23 (Deadlock- and Leak-freedom). $\Gamma \vdash P$ and $P \Rightarrow P'$, then P' neither is deadlocked, nor has a leak.

Corollary 24. Suppose Γ, γ, ℓ : $\langle bool \rangle_{10} \vdash P$, and suppose that the usage of all names in $S = dom(\Gamma, \gamma)$ is 11. Then $(vS)P \downarrow_{\overline{\ell}\backslash b}$ for some b.

Proof. Immediate by Lemma 22 and subject reduction.

This property is used to define barbed equivalence below. It does not hold for higher-order locks: simply discarding x, the lock stored in ℓ , might break typability if ℓ carries an obligation.

3.2 Typed Behavioural Equivalence in $\pi \ell w$

3.2.1 Barbed Equivalence

In barbed equivalence in $\pi\ell$ (Definition 13), we compare complete $\pi\ell$ processes, intuitively to prevent blocked acquire operations from making certain observations impossible. Similarly, in $\pi\ell$ w, we must also make sure that all wait operations in the processes being observed will eventually be fired. For this, we need to make the process complete (in the sense of Lemma 22), and to add restrictions so that wait transitions are fireable.

However, in order to be able to observe some barbs and discriminate processes, we rely on Corollary 24, and allow names to be unrestricted as long as their type is of the form $\langle \mathsf{bool} \rangle_{10}$. This type means that the lock is first order, and that the context has the wait obligation. In such a situation, interactions at ℓ will never be blocked, the whole process is deadlock-free, and eventually reduces to a parallel composition of releases typed with $\langle \mathsf{bool} \rangle_{10}$. Accordingly, we say that a $\pi\ell$ w process P is wait-closed if $\Gamma \vdash P$ and for any $\ell \in \mathsf{dom}(\Gamma)$, $\Gamma(\ell) = \langle \mathsf{bool} \rangle_{10}$.

A typed relation in $\pi \ell$ w is a set of triples (Γ, P, Q) such that $\Gamma \vdash P$ and $\Gamma \vdash Q$, and we write $\Gamma \vdash P \mathcal{R} Q$ for $(\Gamma, P, Q) \in \mathcal{R}$. Barbed equivalence in $\pi \ell$ w is defined like \simeq (Definition 13), restricting observations to wait-closed processes.

Definition 25 (Barbed equivalence in $\pi \ell w$, \simeq_w). A symmetric typed relation \mathcal{R} is a typed barbed bisimulation if $\Gamma \vdash P\mathcal{R}Q$ implies the three following properties:

- 1. whenever P, Q are wait-closed and $P \to P'$, there is Q' s.t. $Q \Rightarrow Q'$ and $\Gamma \vdash P' \mathcal{R} Q'$;
- 2. *if* P, Q *are wait-closed and* $P \downarrow_{\eta}$ *then* $Q \Downarrow_{\eta}$;
- 3. for any E, Γ' s.t. $\Gamma' \vdash E[P]$ and $\Gamma' \vdash E[Q]$, and E[P], E[Q] are wait-closed, we have $\Gamma' \vdash E[P] \mathscr{R} E[Q]$.

Typed barbed equivalence in $\pi \ell w$, written \simeq_w , is the greatest typed barbed bisimulation.

In the second clause above, η can only be of the form $\overline{\ell}\langle b \rangle$, for some boolean value b. Lemma 14 tells us that we could proceed in the same way when defining \simeq .

3.2.2 Typed Transitions for $\pi \ell w$, and Bisimilarity

We now define a LTS for $\pi \ell w$. Transitions for name deallocation are not standard in the π -calculus. To understand how we deal with these, consider $\ell((\ell')).P \mid Q$: this process can do $\xrightarrow{\ell((\nu))}$ only if Q does not use ℓ . Similarly, in $\ell((\ell')).P \mid \ell(x).Q \mid \overline{\ell}\langle \nu \rangle$, the acquire can be fired, and the wait cannot.

Instead of selecting type-allowed transitions among the untyped transitions like in Section 2.3, we give an inductive definition of typed transitions, written $[\Gamma; P] \xrightarrow{\mu} [\Gamma'; P']$. This allows us to use the rules

Figure 3: $\pi \ell w$, Typed LTS. We omit symmetric versions of rules involving parallel compositions

for parallel composition in order to control the absence of a lock, when a lock deallocation is involved. Technically, this is done by refining the definition of the operator to compose typing contexts.

Actions of the LTS are defined as follows: $\mu ::= \ell(v) \mid \overline{\ell}(v) \mid \overline{\ell}(\ell') \mid \tau \mid \ell((v)) \mid \tau/\ell$. Name ℓ plays a particular role in transitions along wait actions $\ell((v))$ and wait synchronisations τ/ℓ : since ℓ is deallocated, we must make sure that it is not used elsewhere in the process. We define $\Gamma_1 \bullet_{\mu} \Gamma_2$ as being equal to $\Gamma_1 \bullet \Gamma_2$, with the additional constraint that $\ell \notin \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)$ when $\mu = \ell((v))$ or $\mu = \tau/\ell$, otherwise $\Gamma_1 \bullet_{\mu} \Gamma_2$ is not defined. The rules defining the LTS are given on Figure 3. We define $\text{fln}(\overline{\ell}(\ell')) = \text{fln}(\tau/\ell) = \{\ell\}$, and $\text{fln}(\ell(v)) = \text{fln}(\ell(v)) = \text{fln}(\overline{\ell}(v)) = \{\ell, v\}$ (with the convention that $\{\ell, v\} = \{\ell\}$ if v is a boolean value).

We comment on the transition rules. Rules TR, TA and TW express the meaning of usages (respectively, 01, 0w and 10). In rule TT, ℓ is deallocated, and the restriction on ℓ is removed. In rules TPT, TPTB we rely on operation $\Gamma_1 \bullet_{\mu} \Gamma_2$ to make sure that ℓ does not appear in both parallel components of the continuation process, and similarly for TPP in the case where μ involves deallocation of ℓ .

Typability is preserved by typed transitions: if $\Gamma \vdash P$ and $[\Gamma; P] \xrightarrow{\mu} [\Gamma'; P']$, then $\Gamma' \vdash P'$. Bisimilarity in $\pi \ell$ w takes into account the additional transitions w.r.t. $\pi \ell$, and is sound for \simeq_w .

Definition 26 (Typed Bisimilarity in $\pi \ell w$, \approx_w). A typed relation \mathcal{R} is a typed bisimulation if $\Gamma \vdash P\mathcal{R}Q$

implies that whenever $[\Gamma; P] \xrightarrow{\mu} [\Gamma'; P']$ *, we have*

- 1. either $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ and $\Gamma' \vdash P' \mathcal{R} Q'$ for some Q'
- 2. or μ is an acquire $\ell(v)$, $Q \mid \overline{\ell}(v) \Rightarrow Q'$ and $\Gamma' \vdash P' \mathcal{R} Q'$ for some Q',
- 3. or μ is a wait $\ell((v))$, $(v\ell)(Q \mid \overline{\ell}(v)) \Rightarrow Q'$ and $\Gamma' \vdash P'\mathcal{R}Q'$ for some Q',
- 4. or $\mu = \tau/\ell$, $(\nu\ell)Q \Rightarrow Q'$ and $\Gamma' \vdash P'\mathcal{R}Q'$ for some Q',

and symmetrically for the typed transitions of Q. Typed bisimilarity in $\pi \ell w$, written \approx_w , is the largest typed bisimulation.

Proposition 27 (Soundness). *For any* Γ , P, Q, *if* $\Gamma \vdash P \approx_w Q$, *then* $\Gamma \vdash P \simeq_w Q$.

```
Example 28. The law \ell(x).\overline{\ell}\langle x\rangle = \mathbf{0} holds in \pi\ell w, at type \ell: \langle \mathsf{T}\rangle_{00}, for any \mathsf{T}. Suppose \Gamma \vdash \ell(x).P \mid \ell((y)).Q. Then we can prove \Gamma \vdash \ell(x).P \mid \ell((y)).Q \approx_w \ell(x).(P \mid \ell((y)).Q). Using this equivalence and the law of asynchrony, we can deduce \ell((x)).P \simeq_w \ell(x).(\overline{\ell}\langle x\rangle \mid \ell((x)).P).
```

An equivalence between $\pi \ell$ processes is also valid in $\pi \ell w$. To state this property, given P in $\pi \ell$, we introduce $[\![P]\!]_w$, its translation in $\pi \ell w$. The definition of $[\![P]\!]_w$ is simple, as we just need to add wait constructs under restrictions for $[\![P]\!]_w$ to be typable.

```
Lemma 29. Suppose \Gamma; \mathbb{R} \vdash P \approx Q. Then \Gamma_w \vdash [\![P]\!]_w \approx_w [\![Q]\!]_w for some \pi \ell w typing environment \Gamma_w.
```

This result shows that the addition of wait does not increase the discriminating power of contexts. We refer to Appendix B for the definition of $[P]_w$ and a discussion of the proof of Lemma 29.

4 Related and Future Work

The basic type discipline for lock names that imposes a safe usage of locks by always releasing a lock after acquiring it is discussed in [13]. This is specified using *channel usages* (not to be confused with the usages of Section 3.1). Channel usages in [13] are processes in a subset of CCS, and can be defined in sophisticated ways to control the behaviour of π -calculus processes. The encoding of references in the asynchronous π -calculus studied in [7] is also close to how locks are used in $\pi \ell w$. A reference is indeed a lock that must be released *immediately* after the acquire. The typed equivalence to reason about reference names in [7] has important differences w.r.t. \simeq_w , notably because the deadlock- and leak-freedom properties are not taken into consideration in that work.

The type system for $\pi \ell$ w has several ideas in common with [12]. That paper studies λ_{lock} , a functional language with higher-order locks and thread spawning. The type system for λ_{lock} guarantees leak- and deadlock-freedom by relying on duality and linearity properties, which entail the absence of cycles. In turn, this approach originates in work on binary session types, and in particular on concurrent versions of the Curry-Howard correspondence [10, 6, 28, 2, 27, 22].

 $\pi\ell$ w allows a less controlled form of interaction than functional languages or binary sessions. Important differences are: names do not have to be used linearly; there is no explicit notion of thread, neither a fork instruction, in $\pi\ell$ w; reduction is not deterministic. The type system for $\pi\ell$ w controls parallel composition to rule out cyclic structures among interacting processes.

The simplicity of the typing rules, and of the proofs of deadlock- and leak-freedom, can be leveraged to develop a theory of typed behavioural equivalence for $\pi\ell$ and $\pi\ell$ w. Soundness of bisimilarity provides a useful tool to establish equivalence results. Proving completeness is not obvious, intuitively because

the constraints imposed by typing prevent us from adapting standard approaches. The way \approx_w is defined should allow us to combine locks with other programming constructs in order to reason about programs featuring locks and, e.g., functions, continuations, and references. Work in this direction will build on [19, 23, 5, 8, 21].

Our proofs of deadlock- and leak-freedom suggest that there is room for a finer analysis of how lock names are used. It is natural to try and extend our type system in order to accept more processes, while keeping the induced behavioural equivalence tractable. A possibility for this is to add *lock groups* [12], with the aim of reaching an expressiveness comparable to the system in [12]. In a given lock group, locks are ordered, which makes it possible to analyse systems having a cyclic topology.

Relying on orders to program with locks is a natural approach, that has been used to define expressive type systems for lock freedom in the π -calculus [11, 13, 20]. In these works, some labelling is associated to channels or to actions on channels, and the typing rules guarantee that it is always possible to define an order, yielding lock-freedom. We plan to study how our type system can be extended with lock groups or ideas from type systems based on orders.

Rule (1) from Section 1 explains in a concise way how the wait operation behaves. Part of the difficulty in Section 3 is in defining a labelled semantics that is compatible with the 'magic' of executing a wait on ℓ only when the restriction can be put on top of the final release of ℓ . We plan to provide a more operational description of deallocation, using, e.g., reference counting as in [12]. $\pi\ell$ w could then be seen as a language to describe at high-level what happens at a lower level when using and deallocating locks.

Acknowledgement. We are grateful to Jules Jacobs for an interesting discussion about this work, and for suggesting the reduction rule (1) from Section 1. We also thank the anonymous referees for their helpful remarks and advices.

References

- [1] Roberto M. Amadio, Ilaria Castellani & Davide Sangiorgi (1998): On Bisimulations for the Asynchronous pi-Calculus. Theor. Comput. Sci. 195(2), pp. 291–324, doi:10.1016/S0304-3975(97)00223-5.
- [2] Luís Caires & Frank Pfenning (2010): Session Types as Intuitionistic Linear Propositions. In Paul Gastin & François Laroussinie, editors: CONCUR 2010 Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings, Lecture Notes in Computer Science 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.
- [3] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): Session types revisited. Inf. Comput. 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.
- [4] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2022): Session Types Revisited: A Decade Later. In: PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 22, 2022, ACM, pp. 12:1–12:4, doi:10.1145/3551357.3556676.
- [5] Adrien Durier, Daniel Hirschkoff & Davide Sangiorgi (2018): Eager Functions as Processes. In Anuj Dawar & Erich Grädel, editors: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, ACM, pp. 364–373, doi:10.1145/3209108.3209152.
- [6] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.
- [7] Daniel Hirschkoff, Enguerrand Prebet & Davide Sangiorgi (2020): On the Representation of References in the Pi-Calculus. In Igor Konnov & Laura Kovács, editors: 31st International Conference on Concurrency Theory, CONCUR 2020, LIPIcs 171, Schloss Dagstuhl Leibniz-Zentrum für Informatik, pp. 34:1–34:20, doi:10.4230/LIPIcs.CONCUR.2020.34.

- [8] Daniel Hirschkoff, Enguerrand Prebet & Davide Sangiorgi (2021): On sequentiality and well-bracketing in the π-calculus. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 July 2, 2021, IEEE, pp. 1–13, doi:10.1109/LICS52264.2021.9470559.
- [9] Kohei Honda & Mario Tokoro (1991): An Object Calculus for Asynchronous Communication. In Pierre America, editor: ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings, Lecture Notes in Computer Science 512, Springer, pp. 133–147, doi:10.1007/BFb0057019.
- [10] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): Language Primitives and Type Discipline for Structured Communication-Based Programming. In Chris Hankin, editor: Programming Languages and Systems ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 April 4, 1998, Proceedings, Lecture Notes in Computer Science 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.
- [11] Atsushi Igarashi & Naoki Kobayashi (2001): A generic type system for the Pi-calculus. In Chris Hankin & Dave Schmidt, editors: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001, ACM, pp. 128–141, doi:10.1145/360204.360215.
- [12] Jules Jacobs & Stephanie Balzer (2023): *Higher-Order Leak and Deadlock Free Locks*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1027–1057, doi:10.1145/3571229.
- [13] Naoki Kobayashi (2002): Type Systems for Concurrent Programs. In Bernhard K. Aichernig & T. S. E. Maibaum, editors: Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers, Lecture Notes in Computer Science 2757, Springer, pp. 439–453, doi:10.1007/978-3-540-40007-3_26.
- [14] Naoki Kobayashi (2007): Type Systems for Concurrent Programs. Extended version of [13].
- [15] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. *ACM Trans. Program. Lang. Syst.* 21(5), pp. 914–947, doi:10.1145/330249.330251.
- [16] Massimo Merro & Davide Sangiorgi (2004): *On asynchrony in name-passing calculi*. *Math. Struct. Comput. Sci.* 14(5), pp. 715–767, doi:10.1017/S0960129504004323.
- [17] R. Milner (1991): *The polyadic π-calculus: a tutorial*. Technical Report ECS–LFCS–91–180, LFCS. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.
- [18] Robin Milner (1980): A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer, doi:10.1007/3-540-10235-3.
- [19] Robin Milner (1992): Functions as Processes. Math. Struct. Comput. Sci. 2(2), pp. 119–141, doi:10.1017/S0960129500001407.
- [20] Luca Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In Thomas A. Henzinger & Dale Miller, editors: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, *CSL-LICS* '14, Vienna, Austria, July 14 18, 2014, ACM, pp. 72:1–72:10, doi:10.1145/2603088.2603116.
- [21] Enguerrand Prebet (2022): Functions and References in the Pi-Calculus: Full Abstraction and Proof Techniques. In Mikolaj Bojanczyk, Emanuela Merelli & David P. Woodruff, editors: 49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France, LIPIcs 229, Schloss Dagstuhl Leibniz-Zentrum für Informatik, pp. 130:1–130:19, doi:10.4230/LIPIcs.ICALP.2022.130.
- [22] Pedro Rocha & Luís Caires (2023): Safe Session-Based Concurrency with Shared Linear State. In Thomas Wies, editor: Programming Languages and Systems 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings, Lecture Notes in Computer Science 13990, Springer, pp. 421–450, doi:10.1007/978-3-031-30044-8_16.

- [23] Davide Sangiorgi (1994): *The Lazy Lambda Calculus in a Concurrency Scenario*. Inf. Comput. 111(1), pp. 120–153, doi:10.1006/inco.1994.1042.
- [24] Davide Sangiorgi (1996): *pi-Calculus, Internal Mobility, and Agent-Passing Calculi.* Theor. Comput. Sci. 167(1&2), pp. 235–274, doi:10.1016/0304-3975(96)00075-8.
- [25] Davide Sangiorgi (1997): The Name Discipline of Uniform Receptiveness (Extended Abstract). In Pierpaolo Degano, Roberto Gorrieri & Alberto Marchetti-Spaccamela, editors: Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings, Lecture Notes in Computer Science 1256, Springer, pp. 303–313, doi:10.1007/3-540-63165-8_187.
- [26] Davide Sangiorgi & David Walker (2001): *The Pi-Calculus a theory of mobile processes*. Cambridge University Press.
- [27] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In Matthias Felleisen & Philippa Gardner, editors: Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Lecture Notes in Computer Science 7792, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.
- [28] Philip Wadler (2014): *Propositions as sessions*. *J. Funct. Program.* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

A Additional Material for Section 2

A.1 CCS ℓ , Operational Semantics

Structural congruence is the least congruence satisfying the following axioms:

$$\overline{P \mid Q \equiv Q \mid P}$$

$$\overline{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}$$

$$\overline{P \mid (v\ell)Q \equiv (v\ell)(P \mid Q)} \text{ if } \ell \notin \text{fln}(P)$$

$$\overline{(v\ell)(v\ell')P \equiv (v\ell')(v\ell)P}$$

To define reduction, we introduce *execution contexts*, E, given by $E ::= [\cdot] \mid E \mid P \mid (v\ell)E$, where $[\cdot]$ is the hole. E[P] is the process obtained by replacing the hole in E with P.

Reduction is defined by the following rules:

$$\frac{P \to P'}{\overline{\ell} \mid \ell.P \to P} \qquad \qquad \frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{E[P] \to E[P']}$$

A.2 CCS ℓ , Properties of the Type System

of Lemma 4. We show by induction on k that $\ell_1.P_1 \mid \ldots \mid \ell_{k-1}.P_{k-1}$ is lock-connected: this holds because for every i, $\ell_i.P_i$ is lock-connected, and because $\ell_i.P_i \stackrel{\ell_i}{\longleftrightarrow} \ell_{i+1}.P_{i+1}$ for all i < k.

Moreover, we know $\ell_{k-1}.P_{k-1} \stackrel{\ell_{k-1}}{\longleftrightarrow} \ell_k.P_k$ and $\ell_k.P_k \stackrel{\ell_k}{\longleftrightarrow} \ell_1.P_1$. So names ℓ_{k-1} and ℓ_k belong to the free names both of $\ell_1.P_1 \mid \ldots \mid \ell_{k-1}.P_{k-1}$ and of $\ell_k.P_k$. By Lemma 3, this prevents $\ell_1.P_1 \mid \ldots \mid \ell_k.P_k$ from being typable.

A.3 $\pi \ell$, Operational Semantics

Structural congruence in $\pi \ell$, written \equiv , is standard, except for the treatment of mismatch. Indeed, the corresponding axiom cannot be used under an acquire prefix.

To handle this, we introduce an auxiliary structural congruence relation, written \equiv_r . Relation \equiv is the smallest equivalence relation that satisfies the axioms for \equiv in CCS ℓ , plus the following ones

PNIL RNIL MAT MIS
$$\frac{P \mid \mathbf{0} \equiv P}{(v\ell)\mathbf{0} \equiv \mathbf{0}} \qquad \frac{[v=v]P_1, P_2 \equiv P_1}{[v=v]P_1, P_2 \equiv P_1} \qquad \frac{[v=v']P_1, P_2 \equiv P_2}{[v=v']P_1, P_2 \equiv P_2} \text{ if } v \neq v'$$

and also the contextual axioms

$$\begin{array}{ccc} \text{CPar} & & \text{CRes} & & \text{CAcq} \\ P \equiv Q & & P \equiv Q & & P \equiv_{\text{r}} Q \\ \hline P \mid T \equiv Q \mid T & & (v\ell)P \equiv (v\ell)Q & & \ell(\ell').P \equiv \ell(\ell').Q \end{array}$$

The last axiom refers to \equiv_r , which is defined like \equiv , except that MIS is omitted and CACQ is replaced by

$$\frac{P \equiv_{r} Q}{\ell(\ell').P \equiv_{r} \ell(\ell').Q}$$

Labelled Semantics for $\pi \ell$. Actions of the LTS are defined by $\mu ::= \ell(\nu) \mid \overline{\ell}\langle \nu \rangle \mid \overline{\ell}(\ell') \mid \tau$. The set of free names of μ is defined by $\operatorname{fln}(\overline{\ell}\langle \nu \rangle) = \operatorname{fln}(\ell(\nu)) = \{\ell, \nu\}$, $\operatorname{fln}(\tau) = \emptyset$ and $\operatorname{fln}(\overline{\ell}(\ell')) = \{\ell\}$. The set of bound names of μ is defined by $\operatorname{bln}(\mu) = \emptyset$, except for $\operatorname{bln}(\overline{\ell}(\ell')) = \{\ell'\}$. The transition rules are the following:

$$\frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{\overline{\ell}\langle v \rangle} \frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{\overline{\ell}\langle v \rangle} \frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{\overline{\ell}\langle v \rangle} \frac{P \xrightarrow{\overline{\mu}} P'}{\overline{\ell}\langle v \rangle} \ell \notin \operatorname{fln}(\mu)$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \operatorname{fln}(Q) \cap \operatorname{bln}(\mu) = \emptyset \qquad \frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \frac{P \xrightarrow{\overline{\ell}\langle v \rangle} P'}{P \mid Q \xrightarrow{\tau} (v \ell') (P' \mid Q')}$$

$$\frac{P_1 \xrightarrow{\mu} P_1'}{[v = v] P_1, P_2 \xrightarrow{\mu} P_1'} \frac{P_2 \xrightarrow{\mu} P_2'}{[v = v'] P_1, P_2 \xrightarrow{\mu} P_2'} v \neq v'$$

B Additional Material from Section 3

B.1 Leak-Freedom in $\pi \ell w$

The proof of Lemma 22 follows the approach of the proof of Lemma 5. An additional difficulty with respect to the latter proof is that release and wait obligations on a given lock need not be explicit in the process, in the sense that they can be stored in another lock.

Proof. We first consider the situation where $\Gamma \vdash P_0$, Γ is complete, and we can write

$$P_0 \equiv (vS)(\prod_i \overline{\ell_i} \langle v_i \rangle \mid \prod_j \ell_j(x_j).P_j \mid \prod_k \ell_k((y_k)).Q_k).$$

We let $P = \prod_i \overline{\ell_i} \langle v_i \rangle \mid \prod_j \ell_j(x_j) . P_j \mid \prod_k \ell_k((y_k)) . Q_k$.

We introduce some terminology to reason about this decomposition. A *prime process* is a process of the form $\overline{\ell}\langle v \rangle$, $\ell(x).P'$ or $\ell((y)).P'$. Here "prime" refers to the fact that such processes cannot be decomposed modulo \equiv . We call *subject* of a prime process the name that occurs in subject position in the topmost prefix of that process: these are the ℓ_i s, the ℓ_i s and the ℓ_k s in the decomposition above.

We make the two following observations. First, for any $\ell \in \operatorname{fln}(P)$, either $\ell \in \operatorname{fln}(P_0)$, or a release of ℓ and a wait on ℓ must be available, by typing. Second, none of the ℓ_i is equal to one of the ℓ_j , since otherwise P_0 could reduce. Moreover, if some ℓ_i is equal to one of the ℓ_k s, then P necessarily contains an acquire on ℓ_i , since otherwise P_0 could reduce by performing a wait transition. In the following, we do not consider the prime processes whose subject is in Γ . Recall that these processes are outputs $\overline{\ell_i}\langle v_i \rangle$ with v_i being either of type bool or $\langle \mathsf{T} \rangle_{00}$.

To derive a contradiction, we show that the subject of every prime process occurs free in another prime process having a different subject. We examine the three forms of prime processes.

• Consider first $\ell_j(x_j)$. P_j . The available *release of* ℓ_j cannot occur at top-level, since otherwise P could reduce. The release cannot be available under an acquire or wait prefix on ℓ_j , by typing and by definition of being available.

The release of ℓ_j may be available in one of the $P_{j'}$ s, or in one of the P_k s occurring under a prefix at some lock name different from ℓ_j . In both cases, ℓ_j occurs in another prime process having a different subject.

If the release on ℓ_j is available neither in the P_j s nor in the P_k s, then there exists another release of the form $\overline{\ell}\langle\ell_j\rangle$ for some ℓ , that does not occur under an acquire on ℓ_j . We remark that ℓ 's usage is of the form 1w, and that $\ell \neq \ell_j$.

Thus, the release of ℓ necessarily occurs in a prime process whose subject is different from ℓ_i .

- Consider now $\ell_k((y_k)).P_k$. As above, we reason about the *release of* ℓ_k . The only difference is that the release of ℓ_k may occur at top-level. If this is the case, then there is necessarily an acquire on ℓ_k , otherwise P could reduce. This acquire cannot occur at top-level, since otherwise P could reduce, by performing a wait transition. Hence, there is a prime process whose subject is different from ℓ_k that contains an acquire on ℓ_k .
- Consider $\overline{\ell_i}\langle v_i \rangle$. We reason about the *wait* on ℓ_i . If the wait on ℓ_i occurs at top-level, then, as above, an acquire on ℓ_i must occur in P, since otherwise P_0 could reduce. That acquire on ℓ_i cannot occur at top-level, since otherwise P could reduce. So in this case ℓ_i occurs in a prime process whose subject is different from ℓ_i .

If the wait on ℓ_i does not occur at top-level, then it can occur in a prime process whose subject is different from ℓ_i : that process cannot start with an acquire on ℓ_i since otherwise P could reduce.

The last possibility is that a subterm of the form $\overline{\ell}\langle\ell_i\rangle$ occurs in some other prime process, and ℓ carries the wait obligation. Reasoning as above, the subject of the prime process cannot be ℓ_i .

We have shown that every prime process in the decomposition above whose subject is ℓ can be connected with a different prime process. Like in the proofs of deadlock-freedom, we obtain a cycle, which is impossible by (the counterpart of) Lemma 4.

B.2 Translating a $\pi \ell$ Process in $\pi \ell w$

If P is a $\pi \ell$ process, $[P]_w$ is its translation into $\pi \ell w$, defined as follows:

$$[[(v\ell)P]]_w = (v\ell)([[P]]_w \mid \ell((x)).\mathbf{0}) \qquad [[\ell(\ell').P]]_w = \ell(\ell').[[P]]_w \qquad [[\overline{\ell}\langle v\rangle]]_w = \overline{\ell}\langle v\rangle$$
$$[[P_1 \mid P_2]]_w = [[P_1]]_w \mid [[P_2]]_w \qquad [[[v=v']P_1, P_2]]_w = [v=v'][[P_1]]_w, [[P_2]]_w$$

To prove Lemma 29, we establish a correspondence between typing in $\pi\ell$ and in $\pi\ell$ w. If $\Gamma; \mathbb{R} \vdash P$, the typing environment to type P seen as a $\pi\ell$ w process is constructed by making sorts explicit, and by assigning usage 10 for name ℓ if $\ell \in \mathbb{R}$, and 00 otherwise. Conversely, if $P \in \pi\ell$ can be typed as a $\pi\ell$ w process with $\Gamma_w \vdash P$, then we can suppose that Γ_w does not contain any usage of the form r1. We recover a $\pi\ell$ typing for P by collecting all names having type usage 10 in \mathbb{R} , and erasing type information in the components of Γ_w , yielding Γ , so that $\Gamma; \mathbb{R} \vdash P$.

This correspondence is extended to a correspondence between transitions, so that a bisimulation relation in $\pi\ell$ is also a bisimulation in $\pi\ell$ w, via the aforementioned translation. To prove the latter property, we rely on the equivalence $\{\{v\}\} \vdash (v\ell)(\overline{\ell}\langle v\rangle \mid \ell((x)).\mathbf{0}) \approx_w \mathbf{0} \text{ in } \pi\ell \text{w}.$