

An Observational Theory of Imperative Concurrent Data Structures in the π -Calculus

Luca Fossati and Kohei Honda

Queen Mary University of London

Abstract. We introduce an observational theory of the asynchronous pi-calculus with a linear-affine type discipline which offers rigorous behavioural characterisations of global progress properties such as non-blockingness and wait-freedom, and illustrate its applicability through the process encoding of well-known imperative concurrent data structures. The typed asynchronous transition and induced bisimilarity precisely capture interactions between client threads and concurrent data structures. The transition is enhanced with fairness and partial failure, leading to concise semantic characterisations of non-blockingness and wait-freedom, without mentioning specific synchronisation primitives nor programming languages. The induced bisimilarity is closed under a large class of typed contexts, enabling compositional reasoning. The theory can then be reflected onto concurrent data structures in programming languages through encoding, allowing process-theoretic reasoning techniques to establish their critical properties.

1 Introduction

This paper studies an observational theory of the asynchronous pi-calculus with a linear-affine type discipline which offers rigorous behavioural characterisations and compositional reasoning principles for global progress properties such as non-blockingness and wait-freedom, independent from specific atomicity primitives such as semaphores and CAS or programming languages. In spite of the extensive algorithmic studies on concurrent data structures in the last decades [28, 10], discussions on these notions still lack a rigorous semantic basis, through which we can accurately and compositionally capture observational (in-)equalities among these data structures, offering rigorous criteria for critical engineering practice such as optimisation and static checking.

As a simple example, suppose we wish to compare several queue implementations, some using locks and others lock-free [22], to be used through a common interface such as *enqueue* and *dequeue* operations [22, 21, 9]. How do these two queues differ in their effects on the *observable* behaviour of client threads, in core functions and in failure resilience? What class of semantics-preserving transformations may affect global progress properties of a lock-free queue? How can we extend these principles for behaviours whose interface goes beyond the standard procedure/method invocation, such as those based on asynchronous message passing? To answer these questions, we need a framework which can rigorously measure the observational effects of concurrent data structures, including global progress properties, in a way independent from individual programming languages.

The present paper introduces an observational theory of global progress based on the π -calculus with linear-affine types, and demonstrates its use through non-trivial reasoning examples. In spite of its tiny syntax, the π -calculus can represent a

wide array of computational phenomena covering both functional, imperative and interactional computation [24, 31, 15], making explicit a subtle interplay among language constructs. Following [1, 29, 15], we use linear-affine types, coming from game semantics and linear logic (cf. [25, 19, 17, 8, 16]), in order to enhance the semantic precision of such representations. The type discipline also plays a key role in our characterisations of global progress properties such as non-blockingness. Typed asynchronous transitions and the induced bisimilarity accurately capture observable behaviour of concurrent data structures as processes. For capturing global progress, the bisimilarity is refined with fairness, leading to clean semantic characterisations of non-blockingness and wait-freedom, applicable to any behaviour representable in the linear-affine π -calculus, without referring to specific synchronisation primitives nor programming languages. The theory can then be reflected onto concurrent data structures in programming languages, enabling the use of process-theoretic reasoning principles to establish their critical properties.

Summary of Contributions We introduce the π -calculus with linear-affine types in §2. Using the calculus, we show the following contributions.

- The asynchronous fair LTS augmented with partial failure, which can accurately capture the behaviour of diverse synchronisation behaviours. The induced bisimilarity is congruent for practically all typed contexts (§3.1).
- A behavioural theory of global progress based on the LTS, giving exact observational characterisations of representative global progress properties [28, §4.5.1], non-blockingness and wait-freedom (§3.2).
- The application of the proposed framework to the analysis of the process encoding of sequential, lock-based and non-blocking queues (§4). Our semantic analysis of a non-blocking queue leads to a uniform understanding of functional correctness (linearizability) and global progress (non-blockingness).

As far as we know, the present work is the first to offer a rigorous observational theory of global progress properties including non-blockingness and wait-freedom, applicable to a large class of concurrent behaviours, through the use of the π -calculus. See §5 for further discussions, including comparisons with related works.

The attached Appendix lists auxiliary notions; [6] contains detailed proofs. Their reading is not essential for understanding our main technical contributions.

2 The π -Calculus with Linear-affine Types

2.1 Processes and Reduction

Processes. We use the asynchronous version of the π -calculus [26, 24] augmented with finite branching/selection. We use the following identifiers: *channels* ($a, b, c, g, h, u, u', \dots$), *value variables* (x, y, \dots), *process variables* (X, Y, \dots), and *values* (v, v', \dots) which are the union of channels, *booleans* $\{\mathbf{tt}, \mathbf{ff}\}$ and *numerals* $\{0, 1, \dots\}$. We write e.g. \vec{x} for a vector. *Processes* (P, Q, \dots) have the following grammar.

$$\begin{aligned}
P ::= & u\&_{i \in I} \{l_i(\vec{x}_i).P_i\} \mid \bar{u} \oplus l(\vec{e}) \mid \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ Q \\
& \mid P|Q \mid (\nu u)P \mid (\mu X(\vec{x}).P)\langle \vec{e} \rangle \mid X\langle \vec{x} \rangle \mid \mathbf{0}
\end{aligned}$$

Above e ranges over *expressions*, which are values, value variables, and arithmetic/logical expressions such as $-e$, e_1+e_2 , $e_1\wedge e_2$, $\neg e$ and $e_1=e_2$. In the grammar,

process $u\&_{i \in I}\{l_i(\vec{x}_i).P_i\}$ is a *branching input* at u , offering non-empty branches, each branch having its *label* l_i , formal parameters \vec{x}_i and continuation P_i . Dually, $\bar{u} \oplus l(\vec{e})$ is an *asynchronous selection message* via u , choosing l and passing \vec{e} after evaluation. Branching and selection are encodable [24] but play a key role in typing [15]. We often call a branching and a selection simply *input* and *output*.

The rest is standard: **if** e **then** P **else** Q is a conditional; $P|Q$ is a parallel composition; and $(\nu u)P$ is a hiding. In recursion $(\mu X(\vec{x}).P)(\vec{e})$, X and \vec{x} bind in their free occurrences in P , and X should occur guarded under input in its body P . Finally $X(\vec{e})$ is an instantiation of a recursion with the vector of parameters \vec{e} .

A singleton branch is often used. Fixing a label for single branching, say l , we write $u(\vec{x}).P$ for $u\&\{l(\vec{x}).P\}$, and $\bar{u}(\vec{e})$ for $\bar{u} \oplus l(\vec{e})$. We use the standard notations such as $\bar{u}(\vec{a})P$ (for $(\nu \vec{a})(\bar{u}(\vec{a})|P)$), \bar{a} and $a.P$ (respectively for $\bar{a}()$ and $a().P$) and $!u(\vec{x}).P$ (for $(\mu X().u(\vec{x}).(P|X\langle \rangle))\langle \rangle$), the standard replication).

Reduction. The reduction relation is defined using the *structural congruence* \equiv , given as the minimal congruence closed under the standard rules [24], augmented with the unfolding of recursion, $(\mu X(\vec{x}).P)(\vec{e}) \equiv P\{(\mu X(\vec{x}).P)/X\}\{\vec{e}/\vec{x}\}$. Then the reduction relation \longrightarrow is defined over closed processes modulo \equiv . In the main reduction rules below, we assume \vec{e} and \vec{x}_j have the same length.

$$u\&_{i \in I}\{l_i(\vec{x}_i).P_i\} | \bar{u} \oplus l_j(\vec{e}) \longrightarrow P_j\{\vec{e}/\vec{x}_j\} \quad (j \in I)$$

Above an input interacts with an output at u , and the former's j -th branch P_j is chosen, with \vec{x}_j instantiated into \vec{e} . The remaining base rules are for the conditional, **if** tt **then** P **else** $Q \longrightarrow P$ and **if** ff **then** P **else** $Q \longrightarrow Q$, and for the evaluation of first-order expressions. We then close the relation under $|$ and ν .

Examples: Simple Concurrent Data Structures. The following processes represent simple concurrent data structures as processes [23]. We use recursive equations [23] for readability, which are easily translated into recursive agents.

$$\begin{aligned} \text{Ref}\langle u, v \rangle &\stackrel{\text{def}}{=} u\&\{\text{read}(z) : \bar{z}\langle v \rangle \mid \text{Ref}\langle u, v \rangle, \text{write}(y, z) : \bar{z} \mid \text{Ref}\langle u, y \rangle\} \\ \text{Ref}^{\text{cas}}\langle u, v \rangle &\stackrel{\text{def}}{=} u\&\left\{ \begin{array}{l} \text{read}(z) : \bar{z}\langle v \rangle \mid \text{Ref}^{\text{cas}}\langle u, v \rangle, \text{write}(y, z) : \bar{z} \mid \text{Ref}^{\text{cas}}\langle u, y \rangle, \\ \text{cas}(x, y, z) : \text{if } x = v \text{ then } (\bar{z}\langle \text{tt} \rangle \mid \text{Ref}^{\text{cas}}\langle u, y \rangle) \text{ else } (\bar{z}\langle \text{ff} \rangle \mid \text{Ref}^{\text{cas}}\langle u, v \rangle) \end{array} \right\} \end{aligned}$$

Above $\text{Ref}\langle u, v \rangle$ is a(n atomic) reference, to which $\text{Ref}^{\text{cas}}\langle u, v \rangle$ further adds the standard cas operation. The following reductions show how a cas operation is represented as a sequence of interactions.

$$\begin{aligned} &\text{Ref}^{\text{cas}}\langle a, 0 \rangle \mid (\nu c)(\bar{a} \oplus \text{cas}\langle 0, 1, c \rangle \mid c(x).P) \\ &\longrightarrow (\nu c)(\text{if } 0 = 0 \text{ then } \bar{c}\langle \text{tt} \rangle \mid \text{Ref}^{\text{cas}}\langle u, 1 \rangle \text{ else } \bar{c}\langle \text{ff} \rangle \mid \text{Ref}^{\text{cas}}\langle u, 0 \rangle \mid c(x).P) \\ &\longrightarrow (\nu c)(\text{if } \text{tt} \text{ then } \bar{c}\langle \text{tt} \rangle \mid \text{Ref}^{\text{cas}}\langle u, 1 \rangle \text{ else } \bar{c}\langle \text{ff} \rangle \mid \text{Ref}^{\text{cas}}\langle u, 0 \rangle \mid c(x).P) \\ &\longrightarrow \text{Ref}^{\text{cas}}\langle u, 1 \rangle \mid P\{\text{tt}/x\} \end{aligned}$$

Note the second and third steps are semantically neutral.

The next two agents represent mutual exclusion in different ways. In the second one, we use the notation **if** $\text{cas}(u, v, w)$ **then** P **else** Q which stands for a CAS's client behaviour: $(\nu c)(\bar{u} \oplus \text{cas}\langle v, w, c \rangle \mid c(x).\text{if } x \text{ then } P \text{ else } Q)$.

$$\begin{aligned} \text{Mtx}\langle u \rangle &\stackrel{\text{def}}{=} u(x).\bar{x}(h)h.\text{Mtx}\langle u \rangle \\ \text{Mtx}^{\text{spin}}\langle u \rangle &\stackrel{\text{def}}{=} (\nu c)(!u(x).\mu X.(\text{if } \text{cas}\langle c, 0, 1 \rangle \text{ then } \bar{x}(h)h.\text{Mtx}^{\text{spin}}\langle u \rangle \text{ else } X) \mid \text{Ref}^{\text{cas}}\langle c, 0 \rangle) \end{aligned}$$

In $\text{Mtx}\langle u \rangle$, a mutual exclusion is realised by making its principal channel u unavailable while being locked [18]. $\text{Mtx}^{\text{spin}}\langle u \rangle$ does the same using a cas and spinning. More complex examples are found in Section 4.

2.2 Types and Typing

We use the linear affine type discipline from [19, 15]. Intuitively, a linear channel is used precisely once, while an affine channel is used at most once. Types are equipped with a notion of duality, which enables semantically precise embeddings of various programming language constructs.

First, the grammar of *types* ($\tau, \tau', \sigma, \sigma', \dots$) is given as:

$$\tau ::= \&_{i \in I}^M l_i(\vec{\tau}_i) \mid \oplus_{i \in I}^M l_i(\vec{\tau}_i) \mid \text{int} \mid \text{bool} \mid \perp$$

where $M \in \{A, L, A*, L*\}$. Types of the form $\&_{i \in I}^M l_i(\vec{\tau}_i)$ are *input types*; those of the form $\oplus_{i \in I}^M l_i(\vec{\tau}_i)$ *output types*. Each of these types carries the vector of types $\vec{\tau}_i$. We use the standard dualisation $\bar{\tau}$, which exchanges input and output. i.e. $\overline{\&_{i \in I}^M l_i(\tau_i)}$ is given as $\oplus_{i \in I}^M l_i(\bar{\tau}_i)$, and conversely, setting $\overline{\text{int}} = \text{int}$ and $\overline{\text{bool}} = \text{bool}$. The \perp indicates “no more composition is possible”, when both input and output are present at an affine/linear channel. Reflecting the process notation for single branching, $\downarrow^M(\vec{\tau})$ and $\uparrow^M(\vec{\tau})$ stand for singly branched input and output types.

Among modalities, the *affine* modality (A) indicates a channel with this mode can be used “at most once”; a *linear* modality (L) “exactly once”; the *unbounded linear* modality ($L*$) means an input end is always available and is shared by an unbounded number of dual outputs, similarly for the *unbounded affine* modality ($A*$) except its input end may become unavailable. We stipulate that $A*$ types cannot carry linear types. We also say τ has the $\downarrow M$ -mode (resp. $\uparrow M$ -mode) when τ is an input (resp. output) type of mode M .

The linear mode L is associated to actions within atomic operations. Then we annotate prefixes and conditionals with the linear mode L . An example of atomic operation containing a conditional is the following annotation of $\text{Ref}^{\text{cas}}\langle u, v \rangle$.

$$\text{Ref}^{\text{cas}}\langle u, v \rangle \stackrel{\text{def}}{=} u \&^{L*} \left\{ \begin{array}{l} \text{read}(z) : \bar{z}^L\langle v \rangle \mid \text{Ref}^{\text{cas}}\langle u, v \rangle, \quad \text{write}(y, z) : z^L \mid \text{Ref}^{\text{cas}}\langle u, y \rangle \\ \text{cas}(x, y, z) : \text{if}^L x = v \text{ then } \bar{z}^L\langle \text{tt} \rangle \mid \text{Ref}^{\text{cas}}\langle u, y \rangle \\ \quad \quad \quad \text{else } \bar{z}^L\langle \text{ff} \rangle \mid \text{Ref}^{\text{cas}}\langle u, v \rangle \end{array} \right\}$$

The annotation above says that, after invocation at u (which is repeatedly available [27, 1]), the process carries out a *series of inevitable* actions, which as a whole form an atomic action. $\text{Ref}\langle u, v \rangle$ is similarly annotated.

The typing rules, which simplify [15] by taking off dependency among linear actions, use these type-annotated processes. An *environment* (Γ, Δ, \dots) is a collection of *type assignments* of the forms $u : \tau$ (mapping a channel/variable to a type) and $X : \vec{\tau}$ (process variable to vector of its argument types), treated as a finite map. A typing judgement has the form $\Gamma \vdash P$ (read “ P has a typing Γ ”).

The details of the typing rules, which follow [15] (summarised in Appendix A), are not important for the subsequent technical development, except for the following properties ensured by the type discipline. Below we say $\Gamma \vdash P$ is *closed* if Γ contains no free value/process variables. *Subject* and *object* are standard [25]. A name is *active* if it occurs as a subject not under any input prefix. A *linear reduction* is a reduction at a L -channel, similarly for *affine reduction*.

Proposition 2.1. *Let $\Gamma \vdash P$ be closed. (1) If $P \longrightarrow P'$ then $\Gamma \vdash P'$. (2) If $\Gamma(c)$ has either $\downarrow L$ or $\downarrow A$ mode, then it is active in P . (3) If $\Gamma(c)$ has either $\uparrow L$ or $\downarrow L^*$ mode, and if c does not occur as an object in P , then there is a sequence of linear reductions $P \longrightarrow^* Q$ such that c is active in Q . (4) If P has a redex at an affine/linear channel, it is a unique redex at that channel. (5) If $P \longrightarrow P_1$ is a linear or affine reduction and $P \longrightarrow P_2 \not\equiv P_1$, then, for some Q , $P_1 \longrightarrow Q$ and $P_2 \longrightarrow Q$, the latter reducing the same linear/affine redex as in $P \longrightarrow P_1$.*

(1) is the standard subject reduction. (2,3) say that, combined with (1), an affine input, an unbounded linear input and a linear input/output are enabled modulo finite linear reductions. (4,5) are the standard properties of affine/linear channels. Simple examples of typed processes follow.

Example 2.2 (typed processes).

1. $u : \&^{L^*} \{ \text{read}(\uparrow^L(\text{nat})), \text{write}(\text{nat} \uparrow^L()) \} \vdash \text{Ref}\langle u, 3 \rangle$.
2. $u : \&^{L^*} \{ \text{read}(\uparrow^L(\text{nat})), \text{write}(\text{nat} \uparrow^L()), \text{cas}(\text{natnat} \uparrow^L(\text{bool})) \} \vdash \text{Ref}^{\text{cas}}\langle u, 0 \rangle$.
3. $u : \downarrow^{A^*}(\uparrow^A(\downarrow^A())) \vdash P$ for $P \in \{ \text{Mtx}\langle u \rangle, \text{Mtx}^{\text{spin}}\langle u \rangle \}$. We can also type $\text{Mtx}^{\text{spin}}\langle u \rangle$ as $u : \downarrow^{L^*}(\uparrow^A(\downarrow^A())) \vdash \text{Mtx}^{\text{spin}}\langle u \rangle$, which is not possible for $\text{Mtx}\langle u \rangle$.

3 A Theory of Global Progress

3.1 Typed Transition and Bisimilarity

This section presents an observational theory of global progress. We start from the labelled transition system (LTS), using the following action labels¹, ℓ, ℓ', \dots :

$$\ell ::= \tau \mid (\nu \vec{c})a\&l(\vec{v}) \mid (\nu \vec{c})a \oplus l(\vec{v})$$

We assume the names in \vec{c} are pairwise distinct and occur in \vec{v} in the same order (as in $(\nu cf)a \oplus l\langle bcdfg \rangle$). If \vec{c} is empty, we omit $(\nu \vec{c})$, writing e.g. $a \oplus l\langle \vec{v} \rangle$. For single-branch value passing, we write $(\nu \vec{a})u\langle \vec{v} \rangle$, $(\nu \vec{a})\bar{a}\langle \vec{v} \rangle$, $a\langle \vec{v} \rangle$ and $\bar{a}\langle \vec{v} \rangle$.

We first define the untyped asynchronous LTS following [13] over untyped processes modulo \equiv , which is generated from the following rules:

$$\text{(Bra)} \quad P \xrightarrow{(\nu \vec{c})a\&l(\vec{v})} P|\bar{a}\oplus l\langle \vec{v} \rangle \quad \text{(Sel)} \quad (\nu \vec{c})(P|\bar{a}\oplus l\langle \vec{v} \rangle) \xrightarrow{(\nu \vec{c})\bar{a}\oplus l(\vec{v})} P$$

In (Bra), no name in \vec{c} should occur in P . We also set $P \xrightarrow{\tau} Q$ iff $P \longrightarrow Q$, and close the relation under $|$ and ν in the standard way [1]. In (Bra), an observer asynchronously sends a message to P , which gets composed with P . Such a message is not processed until a reduction between this output and a complementary input. Symmetrically, a process asynchronously sends a message in (Sel).

Now define the *environment transition* $\Gamma \xrightarrow{l} \Gamma'$ by the following rules:

$$\begin{aligned} \Gamma/\vec{c}, a : \&^{L^*, A^*} \{ l_i(\vec{\tau}_i) \}_{i \in I} &\xrightarrow{(\nu \vec{c})a\&l_j\langle \vec{v}_j \rangle} \Gamma, a : \&^{L^*, A^*} \{ l_i(\vec{\tau}_i) \}_{i \in I} \\ \Gamma/\vec{c}, a : \oplus^{L^*, A^*} \{ l_i(\vec{\tau}_i) \}_{i \in I} &\xrightarrow{(\nu \vec{c})a\oplus l_j\langle \vec{v}_j \rangle} \Gamma, a : \oplus^{L^*, A^*} \{ l_i(\vec{\tau}_i) \}_{i \in I} \\ \Gamma/\vec{c}, a : \&^{L, A} \{ l_i(\vec{\tau}_i) \}_{i \in I} &\xrightarrow{(\nu \vec{c})a\&l_j\langle \vec{v}_j \rangle} \Gamma, a : \perp \\ \Gamma/\vec{c}, a : \oplus^{L, A} \{ l_i(\vec{\tau}_i) \}_{i \in I} &\xrightarrow{(\nu \vec{c})a\oplus l_j\langle \vec{v}_j \rangle} \Gamma \\ &\xrightarrow{\tau} \Gamma \end{aligned}$$

¹ Note that ℓ and l have different fonts.

where we assume, with $\vec{\tau}_i = \tau_{i1}.. \tau_{in_i}$ for each i and $\vec{v}_j = v_{1j}..v_{n_jj}$, that $j \in I$ and $\Gamma \vdash v_i : \tau_{ji}$ for $1 \leq i \leq n_j$ (where $\Gamma \vdash w : \tau$ when either $\Gamma(w) = \tau$ or w is a constant of type τ). Γ/\vec{c} denotes the result of taking off the type assignments for \vec{c} from Γ . The environment transition is deterministic: $\Gamma \xrightarrow{\ell} \Gamma_{1,2}$ implies $\Gamma_1 = \Gamma_2$. We write $\Gamma \vdash \ell$ (read: “ Γ allows ℓ ”) when $\Gamma \xrightarrow{\ell} \Gamma'$ for some Γ' . We write $\text{allowed}(\Gamma)$ for the set of subjects of all transitions allowed by Γ . When $\Gamma \vdash \ell$, the expression $\text{after}(\Gamma, \ell)$ (read: “ Γ after ℓ ”) denotes Γ' s.t. $\Gamma \xrightarrow{\ell} \Gamma'$. We can check that if $\Gamma \vdash P$, $P \xrightarrow{\ell} Q$ and $\Gamma \xrightarrow{\ell} \Delta$ then $\Delta \vdash Q$. This observation leads to:

Definition 3.1 (typed LTS). $\Gamma \vdash P \xrightarrow{\ell} \Gamma' \vdash P' \stackrel{\text{def}}{\iff} P \xrightarrow{\ell} P'$ and $\Gamma \xrightarrow{\ell} \Gamma'$.

Example 3.2. Let $\tau \stackrel{\text{def}}{=} \uparrow^A(\text{int})$, $\tau' \stackrel{\text{def}}{=} \downarrow^{A^*}(\tau)$, $\Gamma \stackrel{\text{def}}{=} a : \tau', c : \tau$ and $P \stackrel{\text{def}}{=} !a(x).\bar{x}\langle 2 \rangle \mid \bar{a}\langle c \rangle$. Then we have $\Gamma \vdash P \xrightarrow{(\nu g)a\langle g \rangle} \Gamma, g : \tau \vdash P|\bar{a}\langle g \rangle$. However $\Gamma \vdash P$ does *not* have a transition $\bar{a}\langle c \rangle$ since $\Gamma \not\vdash \bar{a}\langle c \rangle$.

Below we say a relation over typed processes is *typed* when, whenever it relates two processes, their environments coincide. We write e.g. $\Gamma \vdash PRQ$ when $\Gamma \vdash P$ and $\Gamma \vdash Q$ are related by a typed relation \mathcal{R} .

Definition 3.3 (bisimilarity). A typed relation \mathcal{R} is a *weak bisimulation* or often *bisimulation* when, for each $\Gamma \vdash PRQ$, we have: $P \xrightarrow{\ell} P'$ implies $Q \xRightarrow{\hat{\ell}} Q'$ s.t. $P'\mathcal{R}Q'$, and the symmetric case. The maximum bisimulation is written \approx .

Proposition 3.4. \approx is a typed congruence.

The proof is standard. *From now on, we always assume processes and transitions are well-typed, even when we leave environments implicit, writing e.g. $P \xrightarrow{\ell} Q$.*

3.2 Fairness, Partial Failure and Global Progress.

To capture global progress, our theory incorporate the following two elements:

1. *Fairness* [7, 3, 2], to capture, in interleaving semantics, the semantic requirement that a desirable progress is eventually made assuming each potential activity is given some chance to progress.
2. *Partial failure*, to represent potentially stalling activities and their effects on other non-stalling activities, capturing the robustness principle in the standard notion of global progress.

We first introduce fairness. Let $\Gamma \vdash P$ be closed. A subject a in P is *enabled* if it is the subject of a redex in P ; or a is free and active in P and $\Gamma(a)$ has one of the following modes: $\uparrow A$, $\uparrow L$, $\downarrow L$ and $\uparrow L^*$. For the latter, by Proposition 2.1, the *dual* of each mode is eventually available.

In the following, let Φ, Ψ, \dots range over possibly infinite typed transition sequences, often written $\Phi : \Gamma_1 \vdash P_1 \xrightarrow{\ell_1} \Gamma_2 \vdash P_2 \xrightarrow{\ell_2} \dots$, where we assume newly introduced names are always fresh. We say Φ is *maximal* if it is either infinite or extendible only with non-linear inputs, when finite. We now define fairness, which says that all possible computations should proceed.

Definition 3.5 (fairness). A maximal transition sequence Φ from closed $\Gamma \vdash P$ is *fair* if no subject is infinitely often enabled in Φ .

Example 3.6. Let $P = !a.(\bar{b}|\bar{a})|\bar{a}$ and $Q = \text{Ref}(r, 3)|\bar{r} \oplus \text{read}\langle c \rangle$. Then $P|Q$ has an infinite, and non-fair, transition sequence which *never* involves a reduction in Q . But in a fair sequence, the read in Q surely occurs and an output $\bar{c}\langle 3 \rangle$ will ensue.

The use of strong fairness in Definition 3.5 is not restrictive, as discussed in §5.

A major element of global progress properties is the ability of a thread to progress while some other threads may be stalling. This aspect is captured by augmenting the dynamics with *failing reductions*, through the following two rules.

$$\bar{u} \oplus^M l_j \langle \bar{c} \rangle \longrightarrow \mathbf{0} \quad (M \neq L) \quad \text{if } v \text{ then } P \text{ else } Q \longrightarrow \mathbf{0}$$

We augment the τ -transition accordingly. We call it a (*partial*) *failure* since after it occurs, remaining redexes can still progress. We say that a transition sequence is *non-failing* if it does not contain any failing τ -transition. From now on we shall work with this augmented transition system, unless otherwise stated. Bisimulations continue to denote relations based on transitions without failures.

Definition 3.7 (failing output). Given a transition sequence $\Phi : \Gamma_1 \vdash P_1 \xrightarrow{l_1} \dots \xrightarrow{l_{i-1}} \Gamma_i \vdash P_i \dots$, an affine output at c *fails in* $\Gamma_i \vdash P_i$ (or in Φ), if $c \in \text{allowed}(\Gamma_i)$ but no output at c appears in any transition sequence from $\Gamma_i \vdash P_i$. We write $\text{fail}(\Gamma \vdash P)$ for the set of all subjects of outputs which fail in $\Gamma \vdash P$.

Example 3.8. $\bar{r} \oplus \text{read}(c)c(x).\bar{b}\langle x \rangle$ may fail at two points: it may fail to do *read*, right at the beginning; or it may fail to output via b . In contrast, the read/write/cas operations in $\text{Ref}\langle a, v \rangle$ and $\text{Ref}^{\text{cas}}\langle a, v \rangle$ do not fail due to linearity.

We now define the fundamental global progress properties. Below $\Gamma' \vdash P'$ is a *transition derivative* of $\Gamma \vdash P$ if it can be reached by a sequence of typed transitions from $\Gamma \vdash P$.

Definition 3.9 (Non-Blocking, Wait-Free). Let $\Gamma \vdash P$ be closed.

1. $\Gamma \vdash P$ is *non-blocking* if any maximal non-failing fair sequence Φ from a transition derivative $\Delta \vdash Q$ of $\Gamma \vdash P$, where Φ contains $\Delta' \vdash Q'$ and $\text{allowed}(\Delta') \setminus \text{fail}(\Delta \vdash Q) \neq \emptyset$, is s.t. some affine output occurs in Φ after $\Delta' \vdash Q'$.
2. $\Gamma \vdash P$ is *wait-free* if any maximal non-failing fair sequence Φ from a transition derivative $\Delta \vdash Q$ of $\Gamma \vdash P$, where Φ contains $\Delta' \vdash Q'$ and $c \in \text{allowed}(\Delta') \setminus \text{fail}(\Delta \vdash Q)$, is s.t. some affine output on c occurs in Φ after $\Delta' \vdash Q'$.

If we replace “affine outputs” with “linear outputs”, NB and WF trivially holds since linear outputs never fail *and* will eventually come out, cf. Prop. 2.1 (3).

Definition 3.9 captures the hitherto informally presented notions observationally. Compare it with the following traditional, algorithmic definition (from [28]):

“A data structure is *non-blocking* if it guarantees that *some* process will always be able to complete its pending operation in a finite number of its own steps, regardless of the execution speed of other processes.”

In particular, the key algorithmic notion of “arbitrary speed” is captured observationally by combining fair transitions, which offer an arbitrary amount of *positive* progress, with partial failures, which denote *zero* progress². Definition 3.9 (1) says that, in spite of such varied competing activities, *some* operation can always complete, where we capture the notion of “completion” observationally.

It can be shown that the following property, expressing the *potential* to perform an action in the absence of failures, is implied by non-blockingness.

Definition 3.10 (reliability). $\Gamma \vdash P$ is *reliable* if, for each transition derivative where no failure has occurred, say $\Delta \vdash Q$, if $c \in \text{allowed}(\Delta)$, $\Delta \vdash Q$ admits a sequence of transitions which contains an affine output at c .

Proposition 3.11. *If $\Gamma \vdash P$ is reliable and $\Gamma \vdash P \approx Q$ then $\Gamma \vdash Q$ is also reliable.*

By reliability, if an output does not come out in a transition sequence, it is due to a failure, not because of a functional flaw.

Example 3.12 (Non-Blocking, Wait-Free). Here we only show simple examples. For larger, non-trivial examples, see Section 4.

1. $\text{Ref}\langle u, v \rangle$ with affine typing is both NB and WF, similarly for $\text{Ref}^{\text{cas}}\langle u, v \rangle$.
2. Let $\text{Lck}\langle u \rangle \stackrel{\text{def}}{=} (\nu m)(!u(z).\overline{m}(c)h).\overline{z}|\overline{h} \mid \text{Mtx}\langle m \rangle$, which uses the mutex agent given in Example 2.2 for a (trivial) mutual exclusion. Further let $\text{Lck}^{\text{spin}}\langle u \rangle$ be the same agent except we replace $\text{Mtx}\langle m \rangle$ with $\text{Mtx}^{\text{spin}}\langle m \rangle$. Then neither $\text{Lck}\langle u \rangle$ nor $\text{Lck}^{\text{spin}}\langle u \rangle$ satisfies NB or WF.

To elucidate the status of these global progress properties, let us define their “weak” variants. We say $\Gamma \vdash P$ is *weakly non-blocking* (resp. *weakly wait-free*) if it satisfies the same clause as Definition 3.9 (1) (resp. (2)) above except we consider only non-failing transition sequences. For instance, $\text{Lck}\langle u \rangle$ in Example 3.12 is both WNB and WWF, while $\text{Lck}^{\text{spin}}\langle u \rangle$ is WNB but not WWF. We write **NB** (resp. **WF**, **WNB**, **WWF**) for the set of non-blocking (resp. wait-free, weakly-non-blocking, weakly-wait-free) typed processes. We then observe:

Proposition 3.13. **WF** \subsetneq **NB** \cap **WWF** and **NB** \cup **WWF** \subsetneq **WNB**.

We next identify equivalences and pre-orders that are substitutive and preserve these properties, underpinning the usage of our characterisations. Write $\text{WFT}(\Gamma \vdash P)$ for the set of weak fair traces of P incorporating failures (a *weak* fair trace is the result of abstracting τ from a fair transition sequence). We now define two relations which give us a basic compositional principle. Recall below a bisimulation is considered for non-failing transitions.

Definition 3.14 (fair preorder and bisimulation). (1) A *fair preorder* \mathcal{R} is a weak bisimulation s.t. for each $\Gamma \vdash PRQ'$, we have $\text{WFT}(\Gamma \vdash P) \supseteq \text{WFT}(\Gamma \vdash Q)$. The largest fair preorder is denoted \approx_{fair} . (2) A *fair bisimulation* is a symmetric fair preorder. We write \approx_{fair} for the maximum fair bisimulation.

² The assumption that failures occur only in the first part of the derivation implies that there is only a finite number of them.

Note (1) uses \sqsupseteq . As a motivating example, $\text{Lck}^{\text{spin}}\langle u \rangle$ is \approx -equivalent to and \lesssim_{fair} -lesser than $\text{Lck}\langle u \rangle$: the former has more undue but internally fair traces, cf. [7].

We can now present a basic result for compositional reasoning on global progress. Below (pre-)congruency is considered for typed contexts.

Proposition 3.15 (compositionality). (1) \lesssim_{fair} is a pre-congruence except for the composition at $\downarrow A^*$ and $\uparrow A^*$. Under the same restrictions, \approx_{fair} is a congruence. (2) If $P \lesssim_{\text{fair}} Q$ and P is NB (resp. WF), then so is Q .

The proof uses decomposition of a fair transition through Proposition 2.1 (2,3). The lack of A^* -composition in (1) above is not practically a problem since we can always turn A^* -mode into L^* , e.g. $(\nu c)(!u(x).\bar{c}\langle x \rangle \mid \text{Mtx}\langle c \rangle)$ has the same asynchronous behaviour as $\text{Mtx}\langle u \rangle$ but its u can be typed with the mode L^* .

4 Applications

In this section we apply our observational theory to the semantic analysis of three representative data structures for queues, represented as processes: a sequential queue; a lock-based queue; and Michael-Scott's concurrent queue [22]. The process representations are the result of a compositional encoding of imperative programs (discussed in [6]), which, through the preservation of fair semantics, allow our analysis to be reflected onto the original programming representation.

Queues. To define the three queues we use the agents introduced in Section 2, along with the shortened forms as follows (for reading values from a variable):

$$x \triangleleft \text{read}(\bar{y}).P \stackrel{\text{def}}{=} (\nu c)(\bar{x} \oplus \text{read}\langle c \rangle | c(\bar{y}).P)$$

We use similar shortened forms which may however only read from references a projection of values, for which we use (instead of *read*) some ad-hoc naming such as *getPtr*, *getVal*, etc.

$$\begin{aligned} \text{SQueue}\langle r, \text{curr} \rangle &\stackrel{\text{def}}{=} r \& \{ \text{enqueue}(x, u) : (\text{SQueue}\langle r, \text{curr} \cdot [x] \rangle \mid \bar{u}\langle \rangle), \\ &\quad \text{dequeue}(u) : \text{if}^L (\text{curr} \neq []) \\ &\quad \quad \text{then } (\text{SQueue}\langle r, \text{tl}(\text{curr}) \rangle \mid \bar{u}\langle \text{hd}(\text{curr}) \rangle) \\ &\quad \quad \text{else } \text{SQueue}\langle r, \text{curr} \rangle \mid \bar{u}\langle \text{null} \rangle \} \end{aligned}$$

Fig. 1. π -calculus specification of a queue

Figure 1 gives an intuitive specification of the behaviour of a queue in the π -calculus, assuming list data types [23]. $\text{hd}(\text{list})$ and $\text{tl}(\text{list})$ indicate the first element in *list* and the rest of the list, respectively.

$$\begin{aligned} \text{CQueue}(r) &\stackrel{\text{def}}{=} (\nu h)(\nu t)(\nu s)(\text{CQueue}(r, h, t) \mid \text{Ptr}(h, s, 0) \mid \text{Ptr}(t, s, 0) \mid \text{Node}(s, 0, \text{null})) \\ \text{CQueue}(r, \text{head}, \text{tail}) &\stackrel{\text{def}}{=} r \& \{ \text{enqueue}(x, u) : (\text{CQueue}(r, \text{head}, \text{tail}) \mid P_{\text{enq}}(x, \text{tail})) \\ &\quad \text{dequeue}(u) : (\text{CQueue}(r, \text{head}, \text{tail}) \mid P_{\text{deq}}(\text{head}, \text{tail})) \} \end{aligned}$$

Fig. 2. cas-based Concurrent Queue

The cas-based concurrent queue (Figure 2) uses a linked list with *head* and *tail* pointers. Nodes and pointers have the following shapes, respectively:

$$\text{Node}(r, v, \text{ptr}) \stackrel{\text{def}}{=} \text{Ref}\langle r, \langle v, \text{ptr} \rangle \rangle \quad \text{Ptr}(r, \text{nd}, \text{ctr}) \stackrel{\text{def}}{=} \text{Ref}^{\text{cas}}\langle r, \langle \text{nd}, \text{ctr} \rangle \rangle$$

Here for brevity we allow a pair of values to be stored in a reference (which can be encoded easily). A node contains a value and the reference of a pointer, the latter to retrieve its successor in the list (in an empty node, the successor is a special value null). A pointer contains a node reference and a counter to be incremented at each successful `cas`. To scan the queue we would start from `head`, get to the first (dummy) node, get the pointer reference of the latter and from this get the reference of the second linked node, and so on.

```

 $P_{enq}(x, tail) =$ 
  NullPtr( $nlPtr$ ) | Node( $node, v, nlPtr$ ) |
  ( $\mu X_{tag}(u')$ .
     $tail \triangleleft read(last, ctrT)$ .
     $last \triangleleft getPtr(tPtr)$ .
     $tPtr \triangleleft read(next, ctr)$ .
    if ( $next = null$ ) then
      if cas( $tPtr, \langle next, ctr \rangle, \langle node, ctr + 1 \rangle$ ) then
        if cas( $tail, \langle last, ctrT \rangle, \langle node, ctrT + 1 \rangle$ ) then  $\bar{u}'$  else  $\bar{u}'$ 
        else  $X_{tag}(u')$ 
      else
        if cas( $tail, \langle last, ctrT \rangle, \langle next, ctrT + 1 \rangle$ ) then  $X_{tag}(u')$  else  $X_{tag}(u')$ 
    endif
  ) $\langle u \rangle$ 

```

Fig. 3. Enqueue macro

Figure 3 defines P_{enq} , while P_{deq} is left to the Appendix B. We use the notation `if cas(u, v, w) then P else Q` from Examples in Section 2. The main operational features are appending a node and swinging `head` or `tail` forward. Each is performed by a `cas` operation on a pointer. An operation may also swing `tail` to help global progress (to fulfill others' incomplete actions)³.

$$\begin{aligned}
 LQueue(r) &\stackrel{\text{def}}{=} (\nu u)(\nu h)(\nu t)(\nu s)(LQueue(r, h, t) | Mtx\langle u \rangle | LPtr(h, s) | LPtr(t, s) | LENode(s, 0)) \\
 LQueue(r, h, t, l) &\stackrel{\text{def}}{=} r \& \{ enqueue(v, u) : LQueue(r, h, t, l) | (\bar{l}(g)g(y).P_{enq}^{lck}(v, t, c') | c'.(\bar{y}|u)), \\
 &\quad dequeue(u) : LQueue(r, h, t, l) | (\bar{l}(g)g(y).P_{deq}^{lck}(h, t, c') | c'.(\bar{y}|u)) \}
 \end{aligned}$$

Fig. 4. Lock-based Queue

The lock-based queue (Figure 4) also uses a linked-list structure with head and tail pointers. The key part is lock acquisition through a mutex: the rest is obvious list manipulation ($P_{enq}^{lck}(v, t, c')$ and $P_{deq}^{lck}(h, t, c')$) which we leave to [6]. It has a blocking behaviour: its execution is suspended until the lock is acquired.

Bisimilarity (Functional Correctness). A significant outcome of the asynchronous semantics is that it can show these three queues are, as far as we disregard fairness and failures, semantically indistinguishable (the equivalence reflects, through encoding, how client programs observe differences among these data structures). The proof for each queue uses a simple bisimulation between abstract queue states and transition derivatives of empty queues. Since \approx does not incorporate failures, we only work with failure-free transitions.

³ The original algorithm [22] contained additional checks to speed-up computation.

A queue state $St \in QStates$ has the form $\langle \mathcal{R}; \mathbf{Q}; \mathcal{A} \rangle$, where \mathcal{R} is the set of pending requests at the queue, \mathbf{Q} is the list of values currently in the queue and \mathcal{A} is the set of floating acknowledges s.t.:

- The elements in \mathcal{R} are either $enq(v, g)$ or $deq(g)$, where v is a value, g is a continuation name and the continuation names in \mathcal{R} are a set.
- The list \mathbf{Q} consists of values separated by commas (v_1, \dots, v_n) , as in the other two fields. However, \mathcal{R} and \mathcal{A} are sets where we omit brackets (the order is irrelevant); while \mathbf{Q} is a list (the order is essential).
- The elements in \mathcal{A} are of the form $\bar{g}\langle msg \rangle$, where g is a continuation name, msg is a value and the continuation names in \mathcal{A} are a set.

$$\begin{array}{l}
\langle \mathcal{R}; \mathbf{Q}; \mathcal{A} \rangle \xrightarrow{r \& enqueue(v, g)} \langle \mathcal{R}, enq(v, g); \mathbf{Q}; \mathcal{A} \rangle \quad \langle \mathcal{R}; \mathbf{Q}; \mathcal{A} \rangle \xrightarrow{r \& dequeue(g)} \langle \mathcal{R}, deq(g); \mathbf{Q}; \mathcal{A} \rangle \\
\langle \mathcal{R}, enq(v, g); \mathbf{Q}; \mathcal{A} \rangle \xrightarrow{c(g)} \langle \mathcal{R}; \mathbf{Q}, v; \mathcal{A}, \bar{g}\langle OK \rangle \rangle \quad \langle \mathcal{R}, deq(g); v; \mathbf{Q}; \mathcal{A} \rangle \xrightarrow{c(g)} \langle \mathcal{R}; \mathbf{Q}; \mathcal{A}, \bar{g}\langle v \rangle \rangle \\
\langle \mathcal{R}, deq(g); ; \mathcal{A} \rangle \xrightarrow{c(g)} \langle \mathcal{R}; ; \mathcal{A}, \bar{g}\langle KO \rangle \rangle \quad \langle \mathcal{R}; \mathbf{Q}; \mathcal{A}, \bar{g}\langle v \rangle \rangle \xrightarrow{g!(v)} \langle \mathcal{R}; \mathbf{Q}; \mathcal{A} \rangle \quad St \xrightarrow{\tau} St
\end{array}$$

Fig. 5. LTS for abstract queue states

In Figure 5 we define a LTS on abstract queue states where we also label a special kind of internal transitions (called *committing*), in boldfont and with a channel parameter. Sometimes we write \rightarrow_{NC} to indicate a sequence of non-committing τ transitions.

A committing transition corresponds to the event of committing to perform some operation, whilst other operations may not interfere. This approach is general because in any operation, one can always identify such an event.

Below we outline the reasoning for the concurrent queue, which is most non-trivial. The same general reasoning applies to all three queues. First we identify the committing actions in each queue. Then we define a relation between queues and states and rewrite these transition sequences in parallel into a normal form, making it evident that each related pair of an abstract state and a concrete state are essentially identical.

In the case of the concurrent queue, the committing transition for the enqueue operation is the τ -action induced by the initial output of the *cas* to the *tPtr*, as given in Figure 3, when it's successful; similarly for the dequeue. For formal details see Appendix B. We then define the relation R to be the minimal relation between processes and states by the following induction:

$$(1) \text{ CQueue}(r) R \langle ; ; \rangle \quad (2) \text{ if } (Q R St \wedge Q \xrightarrow{\ell} Q' \wedge St \xrightarrow{\ell} St') \text{ then } Q' R St'$$

The relation R is significant as it is directly a bisimulation and its analysis elucidates the fine operational structures of concurrent queues.

For the proof that R is a bisimulation, we first prove that any process P and state St , such that $P R St$, are reachable by the same sequence of transitions from $\text{CQueue}(r)$ and $\langle ; ; \rangle$, respectively. This allows us to establish a “common history”.

Then the proof consists of two parts. In the first part, we show that any transition $P \xrightarrow{\ell} P'$ can be simulated by St . We do it by cases. Given that input and τ transitions are always enabled in any state, we can focus on committing and

output transitions. Let us consider the case of a committing transition, the output case is similar. Then we observe that the occurrence of $P \xrightarrow{\ell} P'$ becomes possible only after an input transition occurs. But since we established a common history, the input transition must have affected St as well. In particular, St has to have a shape such that $St \xrightarrow{\ell} St'$.

The other part of the proof corresponds to the other direction and follows the same general reasoning, relying on the common history, with two main differences. Firstly, an internal non committing transition does not need to be simulated because it does not affect the state. Second, committing and output transitions from St may not be immediately enabled in P . However, we show that through a sequence of internal non-committing transitions from P we can reach a process P' where the given committing or output transition is enabled. All the details are in Appendix B.

Similar relations can be defined for the other two queues, and by a similar reasoning it can be proved that they are also bisimulations. Then we can state that the three queues are bisimilar.

Proposition 4.1 (Correctness). $\text{CEQueue}(r) \approx \text{SQueue}(r) \approx \text{LEQueue}(r)$.

Without going into the details, we note that the correspondence between queues and states is much tighter than explained. In particular, we can prove that each derivative of the empty queue is related to a unique process (Appendix B). In particular, we can observe that processes and states follow the same dynamics: an input requests some operation, then the queue may commit to perform the requested operation⁴ and then the output acknowledging the operation is sent.

These results and observations allow us to put a derivation in a *normal form*, where operations are “linearised” [11] through a sequence of permutations and where *dispensable* (or effect-less) sequences of transitions are removed. We do not include these constructions in this work because they are not employed in the proofs. However, we note that normal forms may turn out very useful to build powerful verification tools for the global progress properties examined below: by normalising an arbitrary execution we can exhibit its successful behaviour. This shall be definitely a topic for further research.

Global Progress (reliability). Here we briefly outline the results on global progress. Assume given a fair transition sequence with a finite number of failures starting from any derivative. Such a sequence can be added a prefix to form a fair sequence from the initial (empty) concurrent queue:

$$\Phi : \Gamma_Q \vdash \text{CEQueue}(r) \xrightarrow{\ell_1} \Gamma_1 \vdash P_1 \cdots \Gamma_i \vdash P_i \cdots$$

We can then apply essentially the same transformations to this fair transitions with finite failures, so that we obtain two sequences: the initial part which is in normal form and which contains all the finite failures of Φ , reaching $\Gamma_m \vdash P_m$: and the remaining infinite sequence which contains a post-fix of Φ and which does not contain failures. Observe that, when loops are executed by one or more processes,

⁴ For a process this means that the operation shall take place after a sequence of linear reductions.

one of these operation succeeds: the proof is by case analysis of the conditions which lead to looping. Since a failure does not occur, by fairness, we know an output eventually comes out. We conclude:

Proposition 4.2 (NB). $\Gamma_Q \vdash \text{CEQueue}(r)$ *is non-blocking.*

Proposition 4.3 (WWF). $\Gamma_Q \vdash \text{LEQueue}(r)$ *is weakly wait-free.*

We also conjecture these queues are strictly ordered by \approx_{fair} but details are to be examined.

5 Related Work and Further Topics

In the present work, we have presented a formal framework based on the π -calculus, for characterising and analysing key properties of concurrent data structures including global progress; and applied the theory to semantic analysis of different implementation of queues encoded as processes. A key notion used in our behavioural characterisations of global progress is that of fair (maximal) asynchronous transition sequence, whose synchronous version is adapted from [3] to the present asynchronous and typed setting. In this context, the use of strong process typing and induced “enabled actions” (upon which fair sequences rely) would be a major difference. Brookes [2] studies (weak) fairness in asynchronous versions of both *CSP* and *CCS*. A major difference is that the present theory is based on asynchronous π -calculus (which allows infinite buffering) as well as typed transitions with failures. We consider the use of strong fairness in our theory not restrictive because weakly fair execution paths for concurrent programs become strongly fair in their process representations [6].

In [30], process calculi are used to reason about classical synchronisation algorithms. There, the introduction of *fairness* is suggested to capture progress properties though not pursued further. In a more recent work with a similar objective, fairness is used for capturing progress in [4] in PAFAS, a CCS-like process algebra with a timed behaviour. The use of fair timed behaviour allows the authors to capture liveness, applied to verification of Dekker’s algorithm. The main difference of the present work is the use of strongly typed asynchronous fair transitions of the π -calculus augmented with failures, which is necessary for precise characterisations of global progress properties such as non-blockingness and wait-freedom applicable to existing concurrent data structures, which are not found in [4].

Another use of fairness for capturing progress is discussed in [18], which studies refined linear type discipline for the π -calculus and its properties. The work shows reductions at linear channels have progress in fair reduction sequences. Our aim is characterisations and analysis of general behavioural properties, which may not be ensured by a type discipline alone. For example, the concurrent queues discussed in §4 are not typable with known linear type disciplines. The behavioural characterisations for global progress demanded partial failures and strongly typed transitions, which are not treated in [18].

A recent work [5] studies the semantics of Michael-Scott’s queue from the viewpoint of linearisability using action traces. There are common motivations

(substitutability), though the resulting frameworks are quite different. In particular, as shown in §4, the semantic framework given in the present theory offers rigorous behavioural justifications of permutation associated with linearisability, as well as capturing global progress as observational properties. The π -calculus-based reasoning methods (transitions, bisimulations, ...) automatically harness such behaviours as higher-order references.

Further discussions on related work are found in the long version [6].

As further topics, in all the correctness proofs for different kinds of queues reported in §4, we used common reasoning technique, namely the association of state space abstraction and their transitions with concrete transitions. This technique, along with the normalisation of executions through permutations, may also be extended to the use of static and dynamic checking, such as model checking, as well as verification by proof assistants (cf. [30, 32]).

References

1. M. Berger, K. Honda and N. Yoshida. Sequentiality and the π -calculus. Proc. *TLCA'01. LNCS*, 2044. 2001.
2. S. Brookes. Deconstructing CCS and CSP: Asynchronous communication, fairness, and full abstraction. *MFPS 16*. 2000.
3. D. Cacciagrano, F. Corradini and C. Palamidessi. Fair π . Proc. *EXPRESS'06. ENTCS*, 175:3–26. Elsevier Science, 2007.
4. F. Corradini, M. R. Di Berardini and W. Vogler. Liveness of a mutex algorithm in a fair process algebra. *Acta Informatica*, 46(3):209–235. 2009.
5. I. Filipovic, P. W. O'Hearn, N. Rinetzky and H. Yang. Abstraction for concurrent objects. *ESOP'09*. 252–266. 2009.
6. Fossati and Honda. The full version of this paper. In <http://eecs.qmul.ac.uk/~luca/fossacs11/>.
7. N. Francez. *Fairness*. Springer, 1986.
8. J.-Y. Girard. Linear Logic. *TCS*, 50: 1–102. 1987.
9. Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley, 2008.
10. M. Herlihy and B. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2009.
11. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492. 1990.
12. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall. 1985.
13. K. Honda and M. Tokoro. An object calculus for asynchronous communication. Proc. *ECOOP'91*. 1991.
14. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. *ESOP'98*. Springer, 1998.
15. K. Honda and N. Yoshida. A uniform type structure for secure information flow. *POPL'02*. 81–92. ACM Press, 2002.
16. J.M.E. Hyland and C.H.L. Ong. On Full Abstraction for PCF. *Information & Computation*, 163. 285–408. 2000.
17. A. Igarashi and N. Kobayashi, A Generic Type System for the Pi-Calculus. *TCS*, 311, 1-3. 121-163. January, 2004.
18. N. Kobayashi and D. Sangiorgi, A Hybrid Type System for Lock-Freedom of Mobile Processes, ACM TOPLAS. 2010.
19. N. Kobayashi, B.C. Pierce and D.N. Turner, Linearity and the Pi-Calculus, ACM TOPLAS, 21(5). 914–947. September 1999.

20. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7). 558–564. July, 1978.
21. Doug Lea et al. Java Concurrency Package. In <http://gee.cs.oswego.edu/dl>, 2003.
22. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *PODC-15*. 267–275. ACM, 1996.
23. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
24. R. Milner. *Functions as Processes*. In *ICALP'90*, volume 443 of *LNCS*, 167–180. Springer, 1990.
25. R. Milner, The Polyadic π -Calculus: A Tutorial, Logic Algebra of Specification, Marktoberdorf, 1992.
26. R. Milner and J. G. Parrow and D. J. Walker, A calculus of Mobile Processes, *Information and Computation* 100(1), 1–77, 1992.
27. D. Sangiorgi. *icalp97*, The name discipline of uniform receptiveness. *lncs*, 1256. 303–313, Springer, 1997
28. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson–Prentice Hall, 2006.
29. N. Yoshida and M. Berger and K. Honda. Strong Normalisation in the π -Calculus. *Information and Computation*, 191(2004).
30. D. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1(3):273–292, 1989.
31. D. Walker. Objects in the pi-calculus. *Inf. Comput.*, 116(2):253–271, 1995.
32. E. Yahav and M. Sagiv, Automatically verifying concurrent queue algorithms. *ENTCS*, 89. Elsevier, 2003.

$$\begin{array}{c}
\frac{\Gamma(\uparrow A, \uparrow *, \downarrow A^*), \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&_{i \in I}^L l_i(\vec{\tau}_i) \vdash u \&_{i \in I}^L l_i(\vec{x}_i).P_i} \quad \frac{\Gamma(\uparrow *), u : \&_{i \in I}^{L*} l_i(\vec{\tau}_i), \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&_{i \in I}^{L*} l_i(\vec{\tau}_i) \vdash u \&_{i \in I}^{L*} l_i(\vec{x}_i).P_i} \\
\\
\frac{\Gamma(\uparrow A, \uparrow *, \downarrow A^*), \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&_{i \in I}^A l_i(\vec{\tau}_i) \vdash u \&_{i \in I}^A l_i(\vec{x}_i).P_i} \quad \frac{\Gamma(\uparrow *), u : \&_{i \in I}^{A*} l_i(\vec{\tau}_i), \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&_{i \in I}^{A*} l_i(\vec{\tau}_i) \vdash u \&_{i \in I}^{A*} l_i(\vec{x}_i).P_i} \\
\\
\frac{\Gamma \vdash \vec{e} : \vec{\tau}_j \ (j \in I) \quad \text{Dom}(\Gamma) = \text{names}(\vec{e})}{\Gamma, u : \oplus_{i \in I}^M l_i(\vec{\tau}_i) \vdash \bar{u} \oplus^M l_j \langle \vec{e} \rangle} \quad \frac{\Gamma(\uparrow *), \vec{x} : \vec{\tau}, X : \vec{\tau} \vdash P \quad \Gamma \vdash \vec{e} : \vec{\tau}}{\Gamma \vdash (\mu X(\vec{x}).P) \langle \vec{e} \rangle} \quad \frac{\Gamma(\uparrow *) \vdash \vec{e} : \vec{\tau}}{\Gamma, X : \vec{\tau} \vdash X \langle \vec{e} \rangle} \\
\\
\frac{\Gamma(\uparrow A, \uparrow *, \downarrow A^*) \vdash e : \text{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } Q} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \text{if}^L e \text{ then } P \text{ else } Q} \\
\\
\frac{\Gamma_i \vdash P_i (i = 1, 2) \quad \Gamma_1 \succ \Gamma_2}{\Gamma_1 \odot \Gamma_2 \vdash P_1 | P_2} \quad \frac{\Gamma, a : \tau^\perp, \downarrow * \vdash P}{\Gamma \vdash (\nu a)P} \quad \frac{}{\Gamma(\uparrow A, \uparrow *, \downarrow A^*) \vdash \mathbf{0}}
\end{array}$$

Fig. 6. Typing Rules for Processes

A Linear/Affine Typing Rules

We summarise the linear-affine type discipline used in the present inquiry. The construction follows [15, 1, 29] except we do *not* suppress a L -output by a L input. This simplifies the typing rules (e.g. no recording of causality) while ensuring linearity. We also use an L -annotated conditional (used in §3.2). The purpose of the type discipline is to compositionally ensure the properties in Prop.2.1, while keeping enough typability for affine modes.

We assume each type is *input/output alternating*: in an input type $\&_{i \in I}^M l_i(\vec{\tau}_i)$, each τ_i is an output type, and dually. Further A^* -types cannot carry a linear type: and only L^* -types can carry L/A types. The typing rules are given in Figure 6, where “ Γ, Δ ” indicates the union of Γ and Δ as well as demanding their domains are disjoint. “ $\Gamma^{(M_1, \dots, M_n)}$ ” Γ can only contain types of these modes and base types. $*$ stands for both A^* and L^* .

The first two lines are for inputs. A L/A -input can suppress free channels of all “possibly unavailable” modes, i.e. $\uparrow A \uparrow^*$ and $\downarrow A^*$. While $\downarrow L^* / \downarrow L / \downarrow A$ has receptiveness [27], i.e. availability of input channels [1], $\downarrow A^*$ is *not* receptive. A L^* / A^* -input is standard. Thus $\downarrow L, \uparrow L, \downarrow A$ and $\downarrow L^*$ are never suppressed under input. On the next line, the output rule is standard, dualising the carried types $\vec{\tau}_j$ in the premise. The only difference with [15] is that here we pass a vector of expressions, then we need a separate condition to require that Γ types all and only those names appearing in \vec{e} (denoted in the rules by $\text{names}(\vec{e})$). The rules for recursion are also standard. On the next line, the rules for a conditional is standard except, if not L -annotated, a conditional can only “suppress” channels of possibly unavailable modes. On the final line, the rule for $|$ uses the relation \succ and operation \odot from [1, 15]. They are first defined on types, then extended to typings. Intuitively, on a L/A -channel, composition is allowed for at most one input and at most one output; on a L^* / A^* channel, composition is allowed for at most one input and zero or arbitrarily more outputs. Below note \perp is undefined.

1. $\tau \succ \bar{\tau}$ always and $\tau^M \odot \bar{\tau} = \perp$ if $M \in \{A, L\}$, if else $\tau \odot \bar{\tau} = \tau$ if τ is input, and $\tau \odot \bar{\tau} = \bar{\tau}$ if τ is output.
2. $\tau \succ \tau$ if τ has the \uparrow^* -mode or is `int` or `bool` and if so $\tau \odot \tau = \tau$.

3. Otherwise for no $\tau_{1,2}$ we have $\tau_1 \asymp \tau_2$.

Then $\Gamma_1 \asymp \Gamma_2$ if $\Gamma_1(u) \asymp \Gamma_2(u)$ at each common u ; if so, $\Gamma_1 \odot \Gamma_2$ assigns to each $u \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)$ so that $\Gamma(u) = \Gamma_1(u) \odot \Gamma_2(u)$ for each $u \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2)$, otherwise types come from $\Gamma_{1,2}$. The ν -hiding is only possible for a “self-contained” channel, i.e. \perp (input/output are present for L, A) and \downarrow^* (a server type). Finally we allow $\mathbf{0}$ to be typed by possibly unavailable types.

In Prop. 2.1, we demand c does not occur as an object: because of the IO-alternation, this only applies to a $\uparrow L$ -channel. We can avoid such a free channel w.l.o.g. by using *bound name passing* [1, 29] when passing a linear channel.

B Queues

B.1 Concurrent Queues: Dequeue Operation

The dequeue operation is defined similarly to the enqueue one. As follows:

$$\begin{aligned}
P_{deq}(head, tail) &= (\mu X_{tag}(u'). \\
&\quad head \triangleleft read(hNdRef, ctrH). \\
&\quad tail \triangleleft read(tNdRef, ctrT). \\
&\quad hNdRef \triangleleft getPtr(hNdPtr). \\
&\quad hNdPtr \triangleleft getNext(next). \\
&\quad \text{if } (hNdRef = tNdRef) \text{ then} \\
&\quad \quad \text{if } (next = null) \text{ then } \overline{u'}\langle null \rangle \\
&\quad \quad \text{else} \\
&\quad \quad \quad (\nu c)(\overline{tail} \oplus cas\langle\langle tNdRef, ctrT \rangle, \langle next, ctrT + 1 \rangle, c \rangle | c(y). \\
&\quad \quad \quad \text{if } (y = true) \text{ then } X_{tag}\langle u' \rangle \text{ else } X_{tag}\langle u' \rangle) \\
&\quad \text{else } next \triangleleft getVal(x). \\
&\quad \quad (\nu c')(\overline{head} \oplus cas\langle\langle hNdRef, ctrH \rangle, \langle next, ctr + 1 \rangle, c' \rangle | c'(y'). \\
&\quad \quad \text{if } (y' = true) \text{ then} \\
&\quad \quad \quad \overline{u'}\langle x \rangle \\
&\quad \quad \quad \text{else } X_{tag}\langle u' \rangle))\langle u \rangle
\end{aligned}$$

B.2 Concurrent Queues: Committing Transitions

Below we define the committing transitions for the concurrent queue.

Under the *enqueue*(v, u) branch, consider the action

$$\overline{tPtr} \oplus cas\langle\langle next, ctr \rangle, \langle node, ctr + 1 \rangle, c''\rangle$$

the call of the pointer agent over which we shall perform a *cas* operation to add a new node to the queue. If, when we reduce this action, $tPtr$ contains a value equivalent to $\langle next, ctr \rangle$ (the first actual parameter that we pass to *cas*), then we mark the corresponding transition as $\mathbf{c}(u)$. It means that the transition is committing if and only if the enqueue is going to be done, otherwise it is just an ordinary internal transition.

Under the *dequeue* branch, there are not just one but two committing transitions, corresponding to the two possible outcomes that can be returned: *null* if the queue was empty or some value otherwise.

Consider the shortened form $hNdPtr \triangleleft getNxt(next)$, which consists of the following sequence of actions:

$$\overline{hdNdPtr} \oplus read(c) \mid c(g).\bar{g}(h)h(next, nPtr)$$

where the scope of c, g, h and $nPtr$ is limited to the above sequence and to the process they prefix and in particular, $nPtr$ is not going to be used at all. We mark the transition corresponding to the first action, $\overline{hdNdPtr} \oplus read(c)$, as $\mathbf{c}(u)$ if and only if at the time it takes place, $hdNdPtr$ contains a value whose first projection is null (the value that is later going to be stored in $next$).

For the other dequeue committing transition, consider the action:

$$\overline{head} \oplus cas\langle hLink, \langle next, ctr + 1 \rangle, cc \rangle$$

If, when we reduce this action, $head$ contains a value equivalent to $hLink$ (the first actual parameter that we pass to cas), then we mark the corresponding transition as $\mathbf{c}(u)$.

B.3 Proofs of Correctness

Lemma B.1. *Let $Q R St$, then both $CEQueue(r)$ and $\langle ; ; \rangle$ admit a transition sequence such that:*

$$CEQueue(r) \xrightarrow{l_1} \dots \xrightarrow{l_n} Q \quad \langle ; ; \rangle \xrightarrow{l_1} \dots \xrightarrow{l_n} St$$

where the labels l_1, \dots, l_n may be external or committing transitions, but also non-committing τ transitions.

Proof. The base case is $Q = CEQueue(r)$ and $St = \langle ; ; \rangle$. In this case, both $CEQueue(r)$ and $\langle ; ; \rangle$ can reach themselves through the empty sequence.

Otherwise, there have to be P and St_P such that $P R St_P$, $P \xrightarrow{l_P} Q$ and $St_P \xrightarrow{l_P} St$, by definition of R . Then by induction hypothesis,

$$CEQueue(r) \xrightarrow{l_1} \dots \xrightarrow{l_n} P \quad \langle ; ; \rangle \xrightarrow{l_1} \dots \xrightarrow{l_n} St_P$$

which concludes the proof.

Lemma B.2.

$$P R St \wedge P \xrightarrow{\ell} P' \Rightarrow St \xrightarrow{\ell} St'$$

Proof. By Lemma B.1, $CEQueue(r) \xrightarrow{s} P$ and $\langle ; ; \rangle \xrightarrow{s} St$, for some s . Note that input transitions are always enabled in St , as well as internal non-committing ones. Then we need to show that even when $P \xrightarrow{\ell} P'$ is a committing or output transition, it can be simulated by St .

Let $P \xrightarrow{\ell} P'$ be a committing transition: $\ell = \mathbf{c}(g)$, for some continuation name g . Since a committing transition is enabled in P , P has to contain a sub-process which is a derivative of either P_{enq} or P_{deq} and such a derivative can only be there

if a copy of P_{enq} or P_{deq} , respectively, had previously been spawned. But this may only have happened if a selection of *enqueue* or *dequeue*, respectively, had synchronised with the initial branching on r . Such a selection may only have been introduced by an input transition requesting an enqueue or a dequeue operation, respectively. Then s has to contain an input requesting the operation corresponding to $P \xrightarrow{c(g)} P'$.

Without loss of generality, assume this input transition carried a selection to enqueue some value v . Then $\text{CEQueue}(r) \xrightarrow{s} P$ is as follows:

$$\text{CEQueue}(r) \xrightarrow{\ell_1} \dots \xrightarrow{\ell_{i-1}} Q \xrightarrow{r\&enqueue(v,g)} Q' \xrightarrow{\ell_i} \dots \xrightarrow{\ell_n} P$$

where the labels ℓ_1, \dots, ℓ_n may be external or committing transitions, but also non-committing τ transitions. Note however that no committing transition with continuation name g may appear in ℓ_i, \dots, ℓ_n , since such a transition is still enabled at the end of the sequence.

Then also:

$$\langle ; ; \rangle \xrightarrow{\ell_1} \dots \xrightarrow{\ell_{i-1}} r\&enqueue(v,g) \xrightarrow{\ell_i} \dots \xrightarrow{\ell_n} St$$

In particular, since the committing transition for $enq(v, g)$ has not occurred yet the set of pending requests of St contains $enq(v, g)$. Hence $St \xrightarrow{c(g)} St'$, for some state St' .

If $P \xrightarrow{\ell} P'$ is an output on g , its occurrence implies that a previous committing transition with continuation name g appears in s , by a similar reasoning to that of the committing transition's case. Also, as above, no other output on g may appear in s (such a transition would reduce the corresponding output action, disabling $P \xrightarrow{\ell} P'$ forever). Since $\langle ; ; \rangle \xrightarrow{s} St$, the same output has to appear in the set of pending acknowledgements of St . Then $St \xrightarrow{\ell} St'$, for some state St' .

Lemma B.3.

$$P R St \wedge St \xrightarrow{\ell} St' \Rightarrow (P \rightarrow_{NC} P' \xrightarrow{\ell} P'' \vee \ell = \tau)$$

Proof. By Lemma B.1, $\text{CEQueue}(r) \xrightarrow{s} P$ and $\langle ; ; \rangle \xrightarrow{s} St$, for some s . Note that input transitions are always enabled in P . Moreover, τ -transitions do not need to be simulated since they do not affect the state. Then we need to show that even when $St \xrightarrow{\ell} St'$ is a committing or output transition, it can be simulated by P .

Let $St \xrightarrow{c(g)} St'$. By definition, this transition affects a state by removing a request from the pending requests set and changing the queue representation accordingly. Then in St , the set of pending requests must contain a request for the above committing transition. Such a request may only have been added by a previous input transition. Hence this transition has to appear in $\langle ; ; \rangle \xrightarrow{s} St$. Without loss of generality we may assume that the input requested an enqueue, so that s may be decomposed as follows:

$$\langle ; ; \rangle \xrightarrow{s_1} r\&enqueue(v,g) \xrightarrow{s_2} St$$

Then also:

$$\text{CEQueue}(r) \xrightarrow{s_1} Q \xrightarrow{r \& \text{enqueue}(v, g)} Q' \xrightarrow{s_2} P$$

where, in particular, no committing transition on the continuation name g appears in s_2 . Note that the input transition adds a selection $\bar{r} \oplus \text{enqueue}(v, g)$ in parallel to Q . This selection may or may not have been reduced in $Q' \xrightarrow{s_2} P$. Even if that reduction did take place, spawning a copy of P_{enq} , the committing transition for enqueue may not be enabled in P . In order to reach the process where the committing transition with continuation name g is enabled, further reductions of the copy of P_{enq} (and perhaps of other sub-processes as well) are required. We start by assuming that the copy of P_{enq} has not been reduced at all and we show that, under any circumstance, there is always a (possibly empty) sequence of internal transitions from P to some process P' where $\mathbf{c}(g)$ is enabled⁵.

The reduction of the initial selection with the branching on r is enabled any time the selection is there, because any reduction of this branching recursively generates the same process again (indeed, the queue is concurrent). The same also holds for all synchronisations with nodes.

However, synchronising with pointers (i.e. to select *read* or *cas*) may not be as easy, since they are *cas* references, hence they may not be immediately available after a synchronisation. This is the case of the first reduction in P_{enq} , for instance: the selection of *read* on pointer *tail*. This reduction may not be enabled even when we reach a process where the corresponding action is not under any prefix. In particular, s may contain some other reduction selecting *cas* on *tail* but not the reduction of the following conditional (within the body of *cas*). Any time such a situation occurs, we need first to reduce the conditional within the *cas* before our copy of P_{enq} can be reduced further.

Then we get to the first conditional in P_{enq} (condition $\text{next} = \text{null}$) which corresponds to checking that the *tail* pointer points to a node whose successor is *null*. Note that the evaluation of the condition is completely determined by the time we get there since *next* was read in the reduction of the previous action. If it evaluates to *true* then we just proceed towards the committing transition, while if it evaluates to *false* the next step is to try to “swing *tail* forward”. The latter corresponds to the reduction of a *cas* selection on *tail*. For this we pass the pair $\langle \text{last}, \text{ctrT} \rangle$ as first argument, which is compared against the actual values in *tail*. If they match, it means that no other sub-process has updated *tail* since we read it. Then *tail* is updated to $\langle \text{next}, \text{ctrT} + 1 \rangle$ and by a further reduction we go back to the beginning of the loop. Otherwise, somebody else had written *tail* already, then we leave it as it is and we go back to the beginning of the loop. Note that at the next loop, ($\text{next} = \text{null}$) may not evaluate to *false* again, since we are running the simulation and we did not do any synchronisation on $tPtr$.

Then if ($\text{next} = \text{null}$) evaluated to *true*, the next step is a synchronisation for a *cas* on $tPtr$. This could mean we reached the committing transition, if no synchronisation on $tPtr$ occurred after its values were read. Note however that the selection of *read* on $tPtr$ may have occurred as part of $Q' \xrightarrow{s_2} P$ and that by the time P was reached, the contents of $tPtr$ may have been changed. In this

⁵ The same reasoning applies also to the case where a dequeue is selected, reducing P_{deq} .

case, a further reduction would bring us back to the beginning of the loop. Most likely, in the next loop we shall need to swing *tail* forward as well. However, in the third loop, we should eventually reach the committing transition and perform it successfully, since we did not let any other synchronisation on *tPtr* occur in the meantime. Let P' be the derivative of P where the committing transition is enabled. Then note that all the steps we made to get to the event $P' \xrightarrow{c(g)} P''$ were internal non-committing transitions.

Finally, let $St \xrightarrow{\ell} St'$ be an output transition on g . By definition, this transition affects a state by removing an output from the set of floating acknowledges. Then in St , the set of floating acknowledges must contain an output on g . Such an output may only have been added by a previous committing transition with continuation name g . Hence this transition has to appear in $\langle ; ; \rangle \xrightarrow{s} St$, so that s may be decomposed as follows:

$$\langle ; ; \rangle \xrightarrow{s_1} \xrightarrow{c(g)} \xrightarrow{s_2} St$$

Then also:

$$\text{CEQueue}(r) \xrightarrow{s_1} \xrightarrow{c(g)} \xrightarrow{s_2} P$$

where, in particular, no output transition on g appears in s_2 . The occurrence of the above committing transition corresponds to the reduction of (a derivative of) a copy of either P_{enq} or P_{deq} , which may have been further reduced in s_2 but not completely, since the output action corresponding to the considered transition has still to be reduced. Then P has to contain a derivative of this sub-process, where the output transition on g may or may not be enabled. However, if it is not enabled we can further reduce P as we did in the case of the committing transition, so to reach some process P' where it can take place. Then we are done.

Proof of Proposition 4.1 It follows directly from Lemma B.2 and Lemma B.3 that $\text{CEQueue}(r) \approx \langle ; ; \rangle$. We omit the analogous results for the other queues, noting that they follow the same reasoning (and are actually much easier).

B.4 Proofs of Behavioural Properties

Proof of Proposition 4.2 First we show that $\Gamma_Q \vdash \text{CEQueue}(r)$ is reliable. Let there be a sequence of transitions:

$$\Gamma_Q \vdash \text{CEQueue}(r) \xrightarrow{l_1} \dots \xrightarrow{l_p} \Gamma_P \vdash P$$

where Γ_P allows some output *out*. Then *out* shall occur only if we reduce the corresponding action in its thread. Knowing that its thread is a copy of the process which defines the *enqueue* operation, by reducing it we shall eventually but surely reach the *cas* to add the new node at the tail. Note in particular that all the interactions with other nodes and pointers required to get to that point are sure to complete (if one of these agents is involved in an interaction with some other thread we can let this interaction end first). Then the *cas* can be reduced and *out* can occur. $\Gamma_Q \vdash \text{CEQueue}(r)$ is reliable.

Now, let $\Gamma_Q \vdash \text{CEQueue}(r)$ have a transition derivative $\Gamma_P \vdash P$, where Γ_P allows some affine output. Let also Φ be a fair transition derivative from $\Gamma_P \vdash P$. Fairness implies that each thread will progress in Φ until either output, failure or indefinitely. If all threads fail non-blockingness is trivially satisfied, then we consider a thread which does not fail. If the thread takes a branch which exits the loop, it shall output for two reasons: because of the way the `cds` (and the original algorithm) is defined and because we assumed the current thread does not fail. Then the proof boils down to showing that if this thread reduces to itself (recursively) indefinitely in Φ , then eventually some other thread emits an output.

We show it for an enqueue request, the dequeue case is similar. If the condition $y = \text{true}$ evaluates to false, meaning that some node has been added after the node initially pointed to by *tail*, then the thread tries to update the value of *tail* (succeeding if nobody else has already done it). Then at the next loop, it reads the value of *tail* again. If it fails again at the same conditional, it means that somebody has added a new node, since *tail* always points to the last or second to last node in the queue (at the beginning it points to the last one, then the property holds because no node is added if *tail* does not point to the last node, as ensured by the conditional $y = \text{true}$ and by the fact that `cas` is used to modify the last node). If the `cas` on *tPtr* returns *false* (conditional $y' = \text{true}$), it means that the value of the last node's pointer to the next node has changed. Note that this value changes only when a new node is added.

The above reasoning implies that if the current thread reduces to itself indefinitely in Φ (thus making Φ infinite), then infinitely many other threads add each a new node to the queue in Φ . Some of these threads may fail before emitting an output. Since the process is reliable, a thread may only fail by performing a failure reduction. Since we only admit a finite number of failure reductions, an infinite number of threads emits an output in Φ . And this concludes the proof.

Proof of Proposition 4.3 Let $\Gamma_Q \vdash \text{LEQueue}(r)$ have a transition derivative $\Gamma_P \vdash P$, where Γ_P allows some affine output. Let also Φ be a fair transition derivative from $\Gamma_P \vdash P$. Note that the encodings of both *enqueue* and *dequeue* do not contain any conditional, branching (except for singly-branching inputs) or loops. Their reductions consist of a sequence of internal synchronisations that flawlessly reach the final output.

In particular, besides the interactions with c and c' which are introduced by the encoding and which become enabled as they become active, we have interactions with the mutex agent and the actual reductions to perform the operation (enqueue or dequeue). By fairness the mutex agent reduces to its initial state at each interaction. Then an action $\bar{l}(g)$ which is active and attempting to synchronise with the mutex agent, will be cyclically enabled. Then by fairness it shall be reduced. The other actions which synchronise with actions in the mutex agent are immediately reduced, since after the initial synchronisation the agent is accessed exclusively. As for the reductions required to perform the operations of enqueue or dequeue, note that the previous synchronisation with the mutex agent ensures exclusive access to the whole queue, then each action becomes enabled as it becomes active.

Then by fairness, each output allowed by Γ_P shall eventually become active in Φ . Then, also by fairness, it is contained in Φ . In other words, $\text{LEQueue}(r)$ is

weakly wait-free. On the other hand, NB does not hold because a failure in the middle of the critical section would prevent other requests to be processed.

B.5 Proof of the Uniqueness of the State

Here we prove the result we mentioned about the tight relationship between processes and states. But first, the following lemma is required as the proof relies heavily on permutations.

Lemma B.4. *Let $P \xrightarrow{\ell'} P'$ and $P \xrightarrow{\ell''} P''$ be two transitions, where any action reduced by ℓ' has a different subject from any action reduced by ℓ'' . Then there is a process P''' s.t. $P' \xrightarrow{\ell''} P''' \wedge P'' \xrightarrow{\ell'} P'''$.*

Proof. There is only one case where the execution of a transition disables another transition which was previously enabled: that is when the two transitions correspond to different synchronisations of a branching (of which committing transitions are a special case). But this has to be excluded, since we assumed that any action reduced by ℓ' has a different subject from any action reduced by ℓ'' . Moreover, the two transitions must reduce different sub-processes because they reduce two different actions with different subjects, both of which are enabled at the same time. Then reducing one first and then the other, or viceversa, does not affect the final process. We conclude that $P' \xrightarrow{\ell''} P'''$ and $P'' \xrightarrow{\ell'} P'''$, for some process P''' .

Note the statement subsumes the standard partial confluence for a linear/affine reduction as a special case.

Now we are ready for the main proposition. It says that any process reachable from $\text{CEQueue}(r)$ through a sequence of transitions is related to a unique state:

The state St related to P is found by letting $\langle ; ; \rangle$ simulate the sequence of transitions $\text{CEQueue}(r) \xrightarrow{s} P$. The uniqueness of St is proved by taking the last transition in this sequence, $P' \xrightarrow{\ell} P$. Then by cases of this transition it is shown that any derivation from $\text{CEQueue}(r)$ to P can be transformed in another sequence of transitions which ends with $P' \xrightarrow{\ell} P$. The latter sequence is simulated by $\langle ; ; \rangle$, reaching St . Then the inverse transformation is applied which preserves the final state St .

Proposition B.5.

$$\text{CEQueue}(r) \xrightarrow{s} P \Rightarrow \exists! St. P R St$$

Proof. Let $P' R St'$. By Lemma B.1, $\text{CEQueue}(r) \xrightarrow{s} P'$ and $\langle ; ; \rangle \xrightarrow{s} St'$, for some s .

Let $P' \xrightarrow{\ell} P$. If it is an input transition (always enabled), it can always be simulated by St' to obtain St , where a request to the first field is added.

Let $P' \xrightarrow{\ell} P$ be a committing transition. In order for $P' \xrightarrow{\ell} P$ to be enabled, P' must contain a sub-process which is a derivative of either P_{enq} or P_{deq} and such a derivative can only be there if a copy of P_{enq} or P_{deq} , respectively, had previously been spawned. But this may only have happened if a selection of *enqueue* or

dequeue, respectively, had synchronised with the initial branching on r . Such a selection may only have been introduced by an input transition requesting an enqueue or a dequeue operation, respectively. Then s has to contain an input requesting the operation corresponding to $P' \xrightarrow{\ell} P$. Then the same transition has to be enabled also in St' , to obtain St where the request is removed from the request set and the final output on the continuation name is added to the set of pending acknowledgements.

Similarly, if $P' \xrightarrow{\ell} P$ is an output, its occurrence implies a previous committing transition had occurred. Then St' must contain the output in its set of pending acknowledgements, allowing the output transition to occur, after which the output is removed from the set of pending acknowledgements.

Finally, if $\ell = \tau$, it can always be simulated by St' , which is not affected by the transition.

For the uniqueness part, we initially let $P = \text{CEQueue}(r)$. Note that any derivation from $\text{CEQueue}(r)$ starts with an input. Then in order to get back to $\text{CEQueue}(r)$ we need to remove the selection action produced by the input transition, which may only happen through synchronisation. However, such a synchronisation has the effect of spawning a copy of either P_{enq} or P_{deq} . Then we need to reduce this sub-process completely. Note that the reduction of P_{enq} adds other sub-processes (it adds a new node to the queue) and even a successive dequeue leaves some garbage hanging around. Instead, the complete reduction of P_{deq} does lead back to $\text{CEQueue}(r)$ ⁶. However, the simulation of the same derivation by $\langle ; ; \rangle$ brings it also back to itself. Then in this case uniqueness holds.

Now let $\text{CEQueue}(r) \xrightarrow{s} P' \xrightarrow{\ell} P$ such that the state St' satisfying $P' R St'$ is unique. We show by cases of ℓ that P is also related to a unique state. In each case we consider an arbitrary transition sequence from $\text{CEQueue}(r)$ to P and we transform it in a sequence that preserves the initial and final processes ($\text{CEQueue}(r)$ and P , respectively) while having ℓ as last transition. The latter allows us to infer that the antecedent of P in the sequence is P' and that the state St related to P ($P R St$) is also unique⁷ and can be obtained by simulation, $St' \xrightarrow{\ell} St$.

In the following we say that a transition *causally depends* on another one if the former is enabled after the latter occurs but not before. We also say that the two transitions are *causally related*.

1. If ℓ is an input, its occurrence generates a selection action. This may be either an enqueue or a dequeue selection. In any case, it is recognisable from other similar actions which may appear in P , by the continuation name g it contains as last argument. Then any transition sequence from $\text{CEQueue}(r)$ to P has to contain ℓ at some point: $\text{CEQueue}(r) \xrightarrow{t'} \xrightarrow{\ell} \xrightarrow{t''} P$. Note also that only one transition could causally depend on ℓ , the synchronisation of the generated

⁶ Note that the lack of permanent effects in this case, is only due to the fact that the queue was already empty.

⁷ To show its uniqueness, we additionally need to show that the inverse transformation would not change the state.

selection with the branching on r . However, such a synchronisation causes the reduction of both actions and of the selection in particular, which may never be introduced again as we assume (w.l.o.g.) that each new input transition specifies a fresh continuation name. Then no transition in t'' causally depends on ℓ . Moreover, as an input transition, ℓ is always enabled. Then by Lemma B.4 we can permute the original transition sequence to obtain $\text{CEQueue}(r) \xrightarrow{t' t''} P' \xrightarrow{\ell} P$, which can be simulated by the initial state: $\langle ; ; \rangle \xrightarrow{t' t''} St' \xrightarrow{\ell} St$. Since an input transition is always enabled and since permuting it with another transition does not affect the state, the simulation of the original sequence also leads to $St: \langle ; ; \rangle \xrightarrow{t' \ell t''} St$.

2. Let $P' \xrightarrow{\ell} P$ be a committing transition, $\ell = \mathbf{c}(g)$. Then P contains a sub-process which is a derivative of a copy of either P_{enq} or P_{deq} , where the continuation name is g . In particular, this sub-process is the residual of the (local) derivation up to the reduction corresponding to $P' \xrightarrow{\ell} P$. Then the same reduction must appear in every transition sequence from $\text{CEQueue}(r)$ to P , $\text{CEQueue}(r) \xrightarrow{t'} Q' \xrightarrow{\ell} Q'' \xrightarrow{t''} P$. Note also that no further reduction on the same sub-process may appear in $Q'' \xrightarrow{t''} P$, as it would affect the sub-process further.

Now, the transition $Q' \xrightarrow{\ell} Q''$ corresponds to the initial synchronisation on some pointer reference ($tPtr$ or $hdNdPtr$ or $head$) for some operation (\mathbf{cas} or \mathbf{read} or \mathbf{cas} , respectively). Such a synchronisation spawns another sub-process corresponding to the body of the operation, where the continuation name is g and all the parameters have been replaced by actual values. In particular, the value that shall be returned is already determined at this point. Note that $Q' \xrightarrow{\ell} Q''$ is only committing if a certain value is returned (by definition). For the same reasons discussed above for the other sub-process, this sub-process may not be affected in $Q'' \xrightarrow{t''} P$ either. Then, since we know that $P' \xrightarrow{\ell} P$ is committing (in the other derivation), $Q' \xrightarrow{\ell} Q''$ must be committing too.

We also need to check that no other committing transition affects the same pointer in $Q'' \xrightarrow{t''} P$:

- if $Q' \xrightarrow{\ell} Q''$ is an enqueue committing transition, another such transition would need to access $tPtr$, which will not be accessible until the final synchronisation on g ;
- similarly if it is a dequeue committing transition and the queue is not empty (no other \mathbf{cas} on $head$ may be attempted);
- instead in the other case for dequeue (empty queue), a committing transition simply consists of reading a value. Then other such operations may occur, but they do not affect the queue at all. In this case, it can also be shown that no enqueue committing transition may occur because it would imply that the contents of $hdNdPtr$ are different in Q'' , compared to what they are in P . But from the original derivation we can infer that this is false.

Then by Lemma B.4 we can permute the original transition sequence to obtain $\text{CEQueue}(r) \xrightarrow{t' t''} P' \xrightarrow{\ell} P$, which can be simulated by the initial state: $\langle ; ; \rangle \xrightarrow{t' t''} St' \xrightarrow{\ell} St$. From this we can infer, just as in the input case, that the simulation of the original sequence also leads to $St: \langle ; ; \rangle \xrightarrow{t' \ell t''} St$.

3. If ℓ is τ (the label of an internal non-committing transition) we can initially distinguish two cases: whether the transition reduces some action within a `cas` operation or not. In both cases, any transition sequence from $\text{CEQueue}(r)$ to P must contain the same reduction, $\text{CEQueue}(r) \xrightarrow{t'} Q' \xrightarrow{\ell} Q'' \xrightarrow{t''} P$, for the same reasons that were observed in the committing transition case. Moreover, when the transition does not reduce some action within a `cas` the exact same reasoning we did for the input applies.

The only reduction within a `cas` we need to consider is that of the conditional⁸, which may not always be permuted towards the end of the sequence as we did in the previous cases, because some other operation on the same reference may be in the way. In particular, we are not allowed to permute it with the initial synchronisation for the following operation as they are causally related. We shall call *operation A*, the operation to which our reduction belongs, and *Operation B*, the other one. So, by permuting the reduction of the conditional of operation A as far as possible (by applying Lemma B.4), it shall reach the position immediately preceding the synchronisation of operation B. At this point we need to permute the initial synchronisation of operation A towards the end and the reduction of the conditional of operation B towards the beginning (again by Lemma B.4), so to have the two operations occurring in contiguous positions:

$$\text{CEQueue}(r) \xrightarrow{t_1} \xrightarrow{\tau_A} \xrightarrow{\tau_A} \xrightarrow{\tau_B} \xrightarrow{\tau_B} \xrightarrow{t_2} P$$

where τ_A (τ_B) is used to refer to the reductions of operation A (B). We actually need to prove the occurrence in the sequence of the reduction of the conditional of operation B. We observe that the effects of the synchronisation of operation B should be found in P . Then this synchronisation also occurs in the derivation $\text{CEQueue}(r) \xrightarrow{s} P'$. But it could not have occurred after the synchronisation of operation A, because it would have been disabled. Similarly, the synchronisation of operation A would not have been enabled without the reduction of the conditional of operation B. Moreover, since the occurrence of one operation before the other or viceversa does not change the return value (the output action on the continuation name is contained in P and it does not change by doing one derivation or the other), we can swap the two operations at a time:

$$\text{CEQueue}(r) \xrightarrow{t_1} \xrightarrow{\tau_B} \xrightarrow{\tau_B} \xrightarrow{\tau_A} \xrightarrow{\tau_A} \xrightarrow{t_2} P$$

Finally, we permute the reduction of the conditional of operation A towards the end of the sequence, by applying Lemma B.4 again (if other operations

⁸ The initial synchronisation on a `cas` reference has already been discussed for committing transitions, the synchronisation on the continuation name does not cause any complication, just like other internal non-committing transitions.

on the same reference are still in the way, we do again as before, as many times as needed). Note that all these permutations are between an internal non-committing transition and some other transition, then we can infer, just as in the input case, that the simulation of the original sequence also leads to

$$St: \langle ; ; \rangle \xrightarrow{t'} \xrightarrow{\ell} \xrightarrow{t''} St.$$

4. If ℓ is an output, its occurrence reduces the corresponding selection. This prevents us to infer the occurrence of the same transition in any derivation from $CEQueue(r)$ to P . Not only that: by reducing this selection, we are reducing the last action in a previously spawned copy of either P_{enq} or P_{deq} . Note however the same happens to the state which is simulating the process. In particular, once the output occurs, no trace of it remains in the set of pending acknowledges. In the end, the only permanent effects of an operation are the value it may have enqueued or the removal of a value (after a successful dequeue). But, as we have explained, these can also be inferred by looking at the final process (see the case of a committing transition). In the absence of these effects (unsuccessful dequeue), we have already seen that both the process and the state go back to what they were before the operation.

We conclude that there is only one state St such that $P R St$.