Intensional and Extensional Characterisation of Global Progress in the π-Calculus

Luca Fossati, Kohei Honda, and Nobuko Yoshida

Queen Mary University of London and Imperial College London

Abstract. We introduce an observational theory of a linear π-calculus for a rigorous behavioural characterisation of global progress properties such as non-blockingness and wait-freedom. On the basis of typed asynchronous transitions, we introduce a general framework to capture global progress through a combination of fair transitions and partial failures, the latter to represent stalling activities in a fair transition sequence, and show how we can rigorously capture representative global progress properties such as non-blockingness and wait-freedom, both extensionally and intensionally. The intensional characterisations offer a faithful formalisation of the intuitive notions, while the extensional characterisations offer their counterpart capturing a wider class of properties solely based on external observables independent from internal implementations. We introduce a fairness-enriched bisimilarity which preserves these progress properties and which is a congruence, and demonstrate its usage through semantic characterisation and separation results for some of the representative concurrent data structures.

1 Introduction

Imperative concurrent data structures such as lock-based and non-blocking queues play a fundamental role in practice, and have been extensively studied from the algorithmic viewpoint for decades [13, 27]. In spite of these studies, our understanding on some of their key properties such as non-blockingness and wait-freedom still lacks a rigorous semantic foundation, which is essential for critical engineering practice such as verification.

For example, we may wish to compare different queue implementations, some using locks and some lock-free [9, 21, 22] but with a common interface, for substituting a better algorithm for an already existing one. Can we exactly identify their similarities and differences in their observable effects? Can such identifications be extensible to concurrent data structures based on message passing? To answer these questions, we need a uniform semantic theory applicable to a large class of concurrent data structures through which we can accurately analyse their key properties.

Consider the following standard algorithmic description of non-blockingness (also called lock-freedom), taken from [27].

> *"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."*

Here we are considering a data structure which offers a set of operations. When a client requests an operation (say an enqueue), a process/thread is spawned to perform it. When multiple requests arrive, several threads will run concurrently, each of which trying to complete its own operation. The above description says that, in a non-blocking data structure, some of these threads can always complete their operations regardless of how slow or fast other threads are. If we change "some" into "all", we obtain wait-freedom. Although

the description concisely captures the key algorithmic aspects of non-blockingness, it leaves the meaning of its central notions, described in such phrases as "in a finite number of its own steps" and "the execution speed of other processes" informal. It also leaves unspecified whether a run (an execution sequence) being considered can contain unboundedly many requests, hence unboundedly many threads (we shall see the significance of this point in our formal inquiry in § 3). Another issue, from a semantic viewpoint, is that this description is intensional in that it refers to concrete executions of processes (e.g. "its own steps"), which is natural as an algorithmic definition but may not be general enough from a semantic viewpoint, for example for substitutability arguments.

In this paper we introduce a rigorous behavioural theory of concurrent data structures based on a linear π-calculus. The theory can accurately capture not only functional correctness of concurrent data structures but also their global progress properties. The framework uses fair transitions (intuitively because we only wish to consider those runs in which all threads are given sufficient chances to complete) and stalling reductions (formalising the possibility of zero or more threads stopping their execution in a finite or infinite run, in order to measure its effects on other threads, which has been left implicit in standard descriptions such as the above). The framework is uniform in that it can rigorously capture existing global progress properties both intensionally and extensionally. The intensional characterisations offer a faithful formalisation of existing intuitive notions, assigning them a rigorous semantics; whereas the extensional ones capture a wider class of properties solely based on external observables, while being capable of differentiating among typical concurrent data structures.

Another key element of our theory is the use of a series of linear interactions to represent a semantically atomic action: the theory shows how linearity can effectively model the building blocks of diverse synchronisation algorithms. As a result, combined with the expressiveness of the π-calculus, our theory is independent from concrete atomicity primitives such as *semaphore* and *compare-and-swap* (`cas`) or specific programming languages, applicable to a general class of behaviours representable in the π-calculus.

Summary of Contributions. In §**2** we introduce a linear π-calculus. Our main technical contributions include:

- A behavioural theory of asynchronous fair typed labelled transitions augmented with partial failures (stalling), giving rigorous behavioural characterisations of blocking and non-blocking global progress properties in both intensional and extensional settings, leading to their classification (§**3**).
- An application of the proposed framework to the analysis of process encodings of lock-based and non-blocking queues (§**4** and §**5**). A rigorous operational analysis using linear typed transitions leads to a concise proof of semantic linearisability of non-blocking queues (**Th. 5.4**) and semantic separation between a non-blocking queue and a lock-based queue (**Th. 5.9**).

As far as we know, the present work offers the first rigorous complete observational theory of non-blockingness and wait-freedom. See § **7** for further discussions, including comparisons with related works. The Appendix lists auxiliary definitions, further examples and full proofs of all technical results.

2 The π-Calculus with Linear Types

2.1 Processes, Reduction and Types

Processes. Following [15, 29, 33], we use the asynchronous π-calculus augmented with branching/selection. We use the following identifiers: *channels* (or *names*) (a,b,c,g,h, r,u,...); *value variables* ($x, y, \ldots$); *process variables* ($X, Y, \ldots$); *constants* for which we use booleans $\{\mathtt{tt}, \mathtt{ff}\}$ and numerals $\{0, 1, ..\}$; *values* ($v, v', \ldots$) which are the union of channels and constants; and first-order *expressions* ($e, e', \ldots$), inductively generated from values, value variables and first-order operations on them (e.g. $-e$, $e_1{+}e_2$, $e_1{\wedge}e_2$, $\neg e$ and $e_1{=}e_2$).[1] We write $\vec{x}$ ($\vec{e}$) for a vector of distinct variables (expressions).

Processes (P, Q, ...) are given by the following grammar.

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

A *branching* $u\&_{i\in I}\{l_i(\vec{x}_i).P_i\}$ offers non-empty branches, each with a *branch label* l_i, formal parameters $\vec{x}_i$ and continuation P_i. Dually, a *selection* $\overline{u}\oplus l\langle\vec{e}\rangle$ chooses l and passes $\vec{e}$ after evaluation. In each, u occurs as the *subject*, while $\vec{x}_i$ and $\vec{e}$ as *objects*. Branchings and selections are encodable into unary inputs [15], but play a key role in the linear typing we use in this paper. We also use the standard conditional, $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$; parallel composition, $P|Q$; hiding $(\boldsymbol{\nu}\, u)P$ where u binds in its free occurrences in P; recursion $(\mu X(\vec{x}).P)\langle\vec{e}\rangle$, where the initial X and $\vec{x}$ bind in their free occurrences in P, while X in P should occur guarded under an input; and, finally, the instantiation of a recursion $X\langle\vec{e}\rangle$, where $\vec{e}$ is the vector of actual parameters.

Branchings and selections are often called *inputs* and *outputs*, respectively. Without loss of generality, we assume a unique fixed branch label for all single-branch inputs, and omit it in all examples, as in $u(\vec{x}).P$ for input and $\overline{u}\langle\vec{e}\rangle$ for output. We also use the following standard notations: $\overline{u}(\vec{a})P$ (for $(\boldsymbol{\nu}\,\vec{a})(\overline{u}\langle\vec{a}\rangle|P)$), $\overline{a}$ and $a.P$ (for $\overline{a}\langle\rangle$ and $a().P$, respectively) and the replication $!u\&_i\{l_i(\vec{x}_i).P_i\}$ (for $(\mu X().u\&_i\{l_i(\vec{x}_i).(P_i|X\langle\rangle)\})\langle\rangle$).

Reduction. The *structural congruence* $\equiv$ is defined by the standard rules augmented with the unfolding of recursion, $(\mu X(\vec{x}).P)\langle\vec{e}\rangle \;\equiv\; P\{(\mu X(\vec{x}).P)/X\}\{\vec{e}/\vec{x}\}$. The *reduction relation* $\longrightarrow$ is generated from:

$$\begin{array}{c} u\&_{i\in I}\{l_i(\vec{x}_i).P_i\} \mid \overline{u}\oplus l_j\langle\vec{e}\rangle \;\longrightarrow\; P_j\{\vec{v}/\vec{x}_j\} \quad (j\in I,\ \vec{e}\downarrow\vec{v}) \\ \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P \quad (e\downarrow \mathtt{tt}) \qquad \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q \quad (e\downarrow \mathtt{ff}) \end{array}$$

where $\vec{e}\downarrow\vec{v}$ says that the pointwise evaluation of $\vec{e}$ is $\vec{v}$. The first rule says that an input interacts with an output at u, the former's j-th branch P_j is chosen, and $\vec{x}_j$ are instantiated with the evaluation of $\vec{e}$. We close the relation under composition by $|$ and $\boldsymbol{\nu}$, modulo $\equiv$.

Types and Typing. We use the following grammar for types ($\tau, \tau', \sigma, \sigma', \ldots$).

$$\tau \;::=\; \&^{L}_{i\in I} l_i(\vec{\tau}_i) \;\mid\; \&^{\star}_{i\in I} l_i(\vec{\tau}_i) \;\mid\; \oplus^{L}_{i\in I} l_i(\vec{\tau}_i) \;\mid\; \oplus^{\star}_{i\in I} l_i(\vec{\tau}_i) \;\mid\; \bot \;\mid\; \mathsf{bool} \;\mid\; \mathsf{int}$$

In the grammar above, the first four are types for linear/non-linear channels, used for typing branching ($\&$) and selection ($\oplus$) prefixes, where each type in $\vec{\tau}_i$ should not be

[1] In the equality $e_1{=}e_2$, we shall later exclude the name matching by typing.

$\bot$. These types are annotated with *linear* (L) / *non-linear* ($\star$) modes. Linearity is used for representing a single semantically atomic behaviour by a sequence of interactions, as we shall illustrate later. The *dual* of τ (for $\tau \notin \{\bot, \mathsf{bool}, \mathsf{int}\}$), denoted $\overline{\tau}$, is defined by exchanging $\&$ and $\oplus$. We write $\uparrow^L(\vec{\tau})$ for $\oplus^L_1 l_1(\vec{\tau})$ (i.e. singleton). Similarly for $\downarrow^L(\vec{\tau})$, $\uparrow^\star(\vec{\tau})$ and $\downarrow^\star(\vec{\tau})$. Type $\bot$ indicates that both an input and an output are present at a linear channel. bool and int are types for booleans and integers respectively. We define a partial commutative operator $\odot$ generated from: $\tau \odot \overline{\tau} = \bot$ when the modality is linear; $\tau \odot \overline{\tau} = \tau$ with τ a non-linear input; $\tau \odot \tau = \tau$ with τ a non-linear output; and otherwise undefined. In words, composition is allowed for at most one input and at most one output on a linear channel; and, on a non-linear one, it is allowed for at most one input and zero or arbitrarily many outputs.

We use the standard linear typing with branching [15]. A typing environment or simply an *environment* ($\Gamma, \Delta, \ldots$) is a collection of *type assignments*, each of the form $u : \tau$ (channel/variable to type) or $X : \vec{\tau}$ (process variable to a vector of its argument types), forming a finite map. A typing judgement $\Gamma \vdash P$ reads: "P *has typing* Γ". $\Gamma \vdash P$ is *closed* if Γ contains no free value/process variables. Other rules are left to § C.1. The subsequent technical development does not depend on details of typing rules except for the basic properties of typed processes which we shall discuss soon, after introducing the labelled transition.

Linearity Annotation on Terms and Reduction. In a typed process, a *linear channel* is a channel typed with a linear type, ensured to be used exactly once through typing; if not, it is *non-linear*, and may be used zero or more times. A *linear input/output* is an input/output with linear subject. A *linear conditional* is a conditional whose condition may only contain constants, variables and linear channels. Linear inputs, outputs and conditionals are often annotated by a linear mode L, as in $u\&^L_{i\in I}\{l_i(\vec{x}_i).P_i\}$, $\overline{u} \oplus^L l\langle\vec{e}\rangle$ and $\mathtt{if}^L\, v\ \mathtt{then}\ P\ \mathtt{else}\ Q$. We extensively use *linear reduction*, denoted $\longrightarrow_{\mathsf{L}}$, which is a reduction induced by interaction at a linear channel or by reducing a linear conditional.

Processes, Reduction and Types: Examples. An *atomic operation*, such as atomic read-/write and cas, is represented using a sequence of linear reductions. Formally, it consists of an initial *invocation* (i.e. a synchronisation selecting the associated branch) followed by a series of linear reductions, until a response. The channel on which it is invoked is usually non-linear, allowing the operation to be invoked several times. The following example uses recursive equations for readability (easily translatable into recursion).

$$\mathsf{Ref}\langle u, v\rangle \stackrel{\text{def}}{=} u\&\left\{read(z) : \overline{z}^L\langle v\rangle \mid \mathsf{Ref}\langle u, v\rangle,\ write(y, z) : \overline{z}^L \mid \mathsf{Ref}\langle u, y\rangle\right\}$$

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

Above $\mathsf{Ref}\langle u, v\rangle$ represents an atomic reference, to which $\mathsf{Ref}^{\mathsf{cas}}\langle u, v\rangle$ adds the standard cas operation. An example of reduction of the cas atomic operation follows.

$$\begin{array}{l} \mathsf{Ref}^{\mathsf{cas}}\langle a, 0\rangle \mid (\boldsymbol{\nu}\, c)(\overline{a} \oplus cas\langle 0, 1, c\rangle \mid c(x).P) \\ \quad \longrightarrow\ (\boldsymbol{\nu}\, c)((\mathtt{if}\ 0 = 0\ \mathtt{then}\ \overline{c}\langle\mathsf{tt}\rangle \mid \mathsf{Ref}^{\mathsf{cas}}\langle a, 1\rangle\ \mathtt{else}\ \overline{c}\langle\mathsf{ff}\rangle \mid \mathsf{Ref}^{\mathsf{cas}}\langle a, 0\rangle) \mid c(x).P) \\ \quad \longrightarrow_{\mathsf{L}}\ (\boldsymbol{\nu}\, c)(\overline{c}\langle\mathsf{tt}\rangle \mid \mathsf{Ref}^{\mathsf{cas}}\langle a, 1\rangle \mid c(x).P) \\ \quad \longrightarrow_{\mathsf{L}}\ \mathsf{Ref}^{\mathsf{cas}}\langle a, 1\rangle \mid P\{\mathsf{tt}/x\} \end{array}$$

Intuitively, linear reductions inevitably take place regardless of other reductions, because they never get interfered by other actions. Hence we may semantically regard the above sequence as a single atomic action. Later we justify this intuition formally.

As another example, we show two different forms of mutual exclusion. When $\mathsf{Mtx}\langle u\rangle$ gets locked, its principal channel u becomes unavailable [17]; while $\mathsf{Mtx}^{\mathsf{spin}}\langle u\rangle$ uses $\mathtt{cas}$ and spins for representing a locking behaviour.

$$\mathsf{Mtx}\langle u\rangle \stackrel{\text{def}}{=} u(x).\overline{x}(h)h.\mathsf{Mtx}\langle u\rangle$$

$$\mathsf{Mtx}^{\mathsf{spin}}\langle u\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, c)(!u(x).\mu X.(\mathtt{if}\ \mathtt{cas}(c,0,1)\ \mathtt{then}\ \overline{x}(h)h.\mathtt{CAS}(c,1,0)\ \mathtt{else}\ X) \mid \mathsf{Ref}^{\mathsf{cas}}\langle c,0\rangle)$$

Above, the notation "$\mathtt{if}\ \mathtt{cas}(u,v,w)\ \mathtt{then}\ P\ \mathtt{else}\ Q$" stands for the following conditional behaviour, "$(\boldsymbol{\nu}\, c)(\overline{u}\oplus cas\langle v,w,c\rangle|c(x).\mathtt{if}\ x\ \mathtt{then}\ P\ \mathtt{else}\ Q)$"; while the notation "$\mathtt{CAS}(u,v,w)$" stands for doing $\mathtt{cas}$ once, i.e. "$\mathtt{if}\ \mathtt{cas}(u,v,w)\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathbf{0}$".

2.2 Untyped and Typed Labelled Transitions

Untyped Labelled Transition. For our behavioural theory, we use a typed labelled transition system (LTS). The *action labels* $\ell, \ell', \ldots$ are given by the grammar:

$$\ell \quad ::= \quad \boldsymbol{\tau} \mid (\boldsymbol{\nu}\,\vec{c})a\&l(\vec{v}) \mid (\boldsymbol{\nu}\,\vec{c})a\oplus l\langle\vec{v}\rangle$$

Above we assume the names in $\vec{c}$ are pairwise distinct, are disjoint from a, and should occur in $\vec{v}$ in the same order (e.g. $(\boldsymbol{\nu}\, cf)a\oplus l\langle bcdfg\rangle$). If $\vec{c}$ is empty, we omit $(\boldsymbol{\nu}\,\vec{c})$, writing e.g. $a\oplus l\langle\vec{v}\rangle$. For single-branch value passing, we write $(\boldsymbol{\nu}\,\vec{a})u\langle\vec{v}\rangle$, $(\boldsymbol{\nu}\,\vec{a})\overline{a}\langle\vec{v}\rangle$, $a\langle\vec{v}\rangle$ and $\overline{a}\langle\vec{v}\rangle$. Using these action labels, we first define the *untyped* asynchronous LTS. First we set $P \xrightarrow{\boldsymbol{\tau}} Q$ iff $P \longrightarrow Q$. Further we define:

$$\text{(Bra)}\ \ P \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\&l\langle\vec{v}\rangle} P|\overline{a}\oplus l\langle\vec{v}\rangle \qquad \text{(Sel)}\ \ (\boldsymbol{\nu}\,\vec{c})(P|\overline{a}\oplus l\langle\vec{v}\rangle) \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l\langle\vec{v}\rangle} P$$

where, in (Bra), no name in $\vec{c}$ may occur in P. We then close the relation under $\equiv$ as: $P \xrightarrow{\ell} Q$ when $P \equiv P_0$, $P_0 \xrightarrow{\ell} Q_0$ and $Q_0 \equiv Q$. Intuitively, in (Bra), an observer asynchronously sends a message to P which it receives; symmetrically for (Sel).

Environment Transition and Typed Transition. Next we introduce a key technical element of the typed LTS, the transitions of typing environments. $\Gamma \xrightarrow{\ell} \Gamma'$ says that Γ *allows* the action ℓ and the resulting environment is Γ'. First we set $\Gamma \xrightarrow{\boldsymbol{\tau}} \Gamma$, i.e. $\boldsymbol{\tau}$-action is always possible. Further:

$$\begin{aligned}
\Gamma,\ a:\&^{\star}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I} &\xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\&l_j\langle\vec{v}\vec{c}\rangle} \Gamma\odot\vec{v}:\vec{\tau}_j,\ \vec{c}:\vec{\rho}_i,\ a:\&^{\star}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I}\\
\Gamma\odot\vec{v}:\overline{\vec{\tau}_j},\ a:\oplus^{\star}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I} &\xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l_j\langle\vec{v}\vec{c}\rangle} \Gamma,\ \vec{c}:\vec{\rho}_j,\ a:\oplus^{\star}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I}\\
\Gamma,\ a:\&^{L}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I} &\xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\&l_j\langle\vec{v}\vec{c}\rangle} \Gamma\odot\vec{v}:\vec{\tau}_j,\ \vec{c}:\vec{\rho}_j,\ a:\bot\\
\Gamma\odot\vec{v}:\overline{\vec{\tau}_j},\ a:\oplus^{L}\{l_i(\vec{\tau}_i\vec{\rho}_i)\}_{i\in I} &\xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l_j\langle\vec{v}\vec{c}\rangle} \Gamma,\ \vec{c}:\vec{\rho}_j
\end{aligned}$$

Above we separate $\vec{v}$ (free names and constants, typed as $\vec{\tau}_j$ under Γ) and $\vec{c}$ (new names, corresponding to $\vec{\rho}_j$ and disjoint from Γ) for legibility. In the second and fourth rules, $\overline{\vec{\tau}_j}$ is the pointwise dualisation. In the second/fourth rules, if v_i has a non-linear type, then it occurs in Γ. In all rules, $\odot$ should be defined. Intuitively, the first rule says that an

input type assignment to a allows a reception of a message for a and, because of its non-linear type, the typing at a does not change. The second rule is its dual, while the third and fourth rules are their linear variants (only differing in the resulting environments). App. C.1 contains further illustration. The typability of processes and the environment transition are consistent with each other in the following sense.

Proposition 2.1. *If $\Gamma \vdash P$, $\Gamma \xrightarrow{\ell} \Gamma'$ and $P \xrightarrow{\ell} P'$, then $\Gamma' \vdash P'$.*

Proof. See Appendix E.1 to E.2. □

We then set:

$$\Gamma \vdash P \xrightarrow{\ell} \Gamma' \vdash P' \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \Gamma \vdash P,\ P \xrightarrow{\ell} P' \text{ and } \Gamma \xrightarrow{\ell} \Gamma'.$$

By Proposition 2.1, $\Gamma \vdash P \xrightarrow{\ell} \Gamma' \vdash P'$ implies $\Gamma' \vdash P'$.

The following example highlights how a typing controls a typed labelled transition.

Example 2.2 (typed asynchronous transition). Let $\tau \stackrel{\text{def}}{=} \uparrow^{L}(\mathsf{int})$, $\tau' \stackrel{\text{def}}{=} \downarrow^{\star}(\tau)$ and $\Gamma \stackrel{\text{def}}{=} a\colon\tau', c\colon\tau$. Let $P \stackrel{\text{def}}{=} !a(x).\overline{x}\langle 2\rangle \mid \overline{a}\langle c\rangle.$. Then we have:

$$\Gamma \vdash P \xrightarrow{(\boldsymbol{\nu}\, g)a\langle g\rangle} \Gamma, g\colon\tau \vdash P|\overline{a}\langle g\rangle.$$

However $\Gamma \vdash P$ does *not* have a transition $\overline{a}\langle c\rangle$ since $\Gamma \stackrel{\overline{a}\langle c\rangle}{\not\longrightarrow}$. Intuitively, this message should be consumed by the unique input $!a(x).\overline{x}\langle 2\rangle$.

Henceforth we assume processes and transitions are typed, even when we leave environments implicit, as in $P \xrightarrow{\ell} Q$. We use the standard notation $P \stackrel{\hat{\ell}}{\Longrightarrow} Q$ standing for $P \xrightarrow{\boldsymbol{\tau}}{}^{*} Q$ when $\ell = \boldsymbol{\tau}$ and $P \xrightarrow{\boldsymbol{\tau}}{}^{*} \xrightarrow{\ell} \xrightarrow{\boldsymbol{\tau}}{}^{*} Q$ otherwise. $P \xrightarrow{s} P'$ stands for $P \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_n} P'$, where $s = \ell_1 \ldots \ell_n$, then P' is a *transition derivative of* P.

Proposition 2.3. *Below we assume all processes are typed.*

(1) (partial confluence) *Suppose $P \xrightarrow{\ell} Q_1$ and $P \longrightarrow_{\mathsf{L}} Q_2$ s.t. $Q_1 \not\equiv Q_2$. Then there is R such that $Q_1 \longrightarrow_{\mathsf{L}} R$ and $Q_2 \xrightarrow{\ell} R$.*

(2) (linear normal form) *For any P, we have $P \longrightarrow_{\mathsf{L}}^{*} Q \not\longrightarrow_{\mathsf{L}}$ for a unique Q.*

(3) (asynchrony) *$P \xrightarrow{s}\xrightarrow{\ell} Q$ s.t. the subject of input ℓ is not bound in s, implies $P \xrightarrow{\ell}\xrightarrow{s} Q$. And given a free output ℓ, $P \xrightarrow{\ell}\xrightarrow{s} Q$ implies $P \xrightarrow{s}\xrightarrow{\ell} Q$.*

Proof. See Appendix E.3. □

We are now ready to introduce a typed bisimulation. A relation $\mathcal{R}$ over typed processes is *typed* if it relates $\Gamma \vdash P$ and $\Delta \vdash Q$ only if $\Gamma = \Delta$, in which case we write $\Gamma \vdash P\mathcal{R}Q$.

Definition 2.4 (bisimilarity). A relation $\mathcal{R}$ over closed terms is a *(weak) bisimulation* if $P\mathcal{R}Q$ and $P \xrightarrow{\ell} P'$ imply $Q \stackrel{\hat{\ell}}{\Longrightarrow} Q'$ such that $P'\mathcal{R}Q'$, and symmetrically. The largest bisimulation is extended to open terms in the standard way, denoted $\approx$.

Proposition 2.5. *(1) $\approx$ is a typed congruence. (2) $\longrightarrow_{\mathsf{L}} \subset \approx$.*

Proof. See Appendix E.4. □

By Prop. 2.3 (1) and Prop. 2.5 (2), linear reductions are semantically neutral. Further by Prop. 2.3 (2) any transition can be completed by consuming all potential linear reductions at that time. This is why a reduction sequence like the one for $\mathsf{Ref}^{\mathsf{cas}}\langle u, v\rangle$ given in the example at the end of § 2.1, which contains a single non-linear reduction and one or more linear reductions, may be considered as a single semantic action.

3 An Observational Theory of Global Progress

To motivate the semantic setting we introduce in this section, consider wait-freedom. A distinguishing feature of a wait-free data structure is that it ensures the completion of *every* requested operation, even if infinitely many requests arrive (in fact, as our formalisation in this section clarifies and as actual data structures attest, for finitely many requests, non-blockingness also ensures that every operation completes).

Now consider encoding a wait-free queue as a π-calculus process, say P. A run (execution sequence) of this queue, when invoked by one or more requests, corresponds to a labelled transition sequence starting from P, where requests and answers correspond to input and output transitions, respectively. Starting from P, we can easily find infinite runs where some, or even all, of the requests are not answered. An example is a transition sequence that contains only inputs, describing the run where new requests keep coming but none of the requested operations makes progress.

To avoid such an anomaly, we consider only *fair* runs, where every active operation is ensured to progress eventually, i.e. fair transition sequences where no redex is waiting forever. It is well-known that any type of scheduler (sequential, parallel, or a mix of the two) can easily realise fair runs, which are a natural abstraction for many practical implementations. But to our knowledge, no previous algorithmic description of wait-freedom/non-blockingness mentions fairness *explicitly*, although some notion of fairness must be present in the underlying model since runs like the above are *implicitly* excluded.

Explicitly including fairness means each thread can make some progress as far as it continues to get enabled. This however causes the problem of representing the *stalling* of a thread in a run. Consider a fair run of a lock-based queue, where concurrent operations are protected by a lock. Fairness implies that *each* thread entering the critical section will eventually exit (and complete), preventing us from capturing one of the central features of lock-based implementations, i.e. a single thread stalling inside a critical section can block all of the remaining threads whose operations are pending. We can however get an accurate modelling by allowing to arbitrarily reduce an output action to the inaction $\mathbf{0}$ (thus making the corresponding thread stall forever). With this combination of fairness and stalling, our model can accurately represent and differentiate a wide range of global progress properties, both extensionally and intensionally.

3.1 Fair and Failing Sequences

Fairness. We proceed to formally define fair transition sequences. We use the standard strong fairness, which is not restrictive because strongly fair transition sequences in π-calculus encodings correspond to weakly fair runs in concurrent programs. This is due to the fact that the π-calculus offers a more fined-grained representation. For example, in a concurrent program, a request waiting forever to be served would be always enabled,

without discontinuity. This is encoded in the π-calculus as an output action which is infinitely often enabled (as defined below) by the recursive (re-)appearance of the dual input (where each active output, also defined below, can be associated to a unique thread).

We define the notion of *enabledness* as follows (where we say that a channel is active if it occurs as subject not under an input or conditional): 1) a conditional is enabled if it can reduce; 2) an output is enabled when it either can reduce or has a free active subject; 3) a linear input is enabled if it has a free active subject (cf. Prop. 2.3).

Henceforth, $\Phi, \Psi, \ldots$ range over possibly infinite typed transition sequences from closed processes, often written as $\Phi : P_1 \xrightarrow{\ell_1} P_2 \xrightarrow{\ell_2} \cdots$ (omitting environments). We say that a transition sequence Φ is *maximal* if it is either infinite or ends with a process in which no occurrence of conditional, output or linear input is *enabled*. Note finally that we identify an occurrence across transitions (a rigorous treatment is found in [4] which uses labels enriched with occurrences).

Now we define fairness:

Definition 3.1 (fairness). A transition sequence Φ is *fair* if Φ is maximal and no single occurrence of conditional, output or linear input is infinitely often enabled in Φ.

Above we exclude non-linear inputs because we cannot expect the context to surely send a message. On the other hand, a linear input should be inevitable in a fair transition.

Example 3.2 (fairness). In $(!a.\overline{a})|\overline{a} \mid (!b.\overline{b})|\overline{b}$, a non-fair transition sequence is generated when we continuously reduce the a redex, because a single occurrence of $\overline{b}$ is enabled infinitely often. A fair sequence is generated when we alternate the reductions on a and b, because each output occurrence is enabled exactly twice and no more, before the reduction on the other side and before its own.

Failing Reduction and Blocking. As observed already, we capture stalling by adding the following reductions which we call *failing transitions* or *failures*, since we are in effect representing stalling by a non-Bizantine partial failure. Below we assume u is *not* linear and P and Q do not contain linear names.

$$\overline{u} \oplus l_j\langle \vec{e} \rangle \longrightarrow \mathbf{0} \qquad \texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ Q \longrightarrow \mathbf{0}$$

We do not let linear selections/conditionals fail since they are used to denote *atomic operations* (defined in § 2). A transition sequence containing a failing transition is said to be *failing*. It is *finitely failing* if the number of failing reductions is finite.

The purpose of introducing failures is to observe their effects on later transitions: i.e. we wish to observationally capture the notion of a failure in one component blocking other components. This will later allow us to define a key notion in our theory, *resilience*. Resilience itself is not a progress property (it gives a weaker notion of liveness). However it serves as a basis for defining diverse global progress properties uniformly.

Let Φ be a transition sequence which starts from an initial configuration where no requests have yet taken place. We define $\texttt{ended}(\Phi, Q_i)$ as the set of the subjects of the outputs appearing in Φ *before* the occurrence of a process Q_i contained in Φ (to be exact, in $\texttt{ended}(\Phi, Q_i)$, the second element Q_i should be an occurrence of a process in Φ rather than a process itself: this can however be disambiguated by adding e.g. additional inactions to processes (similarly in the next definition). Intuitively, $\texttt{ended}(\Phi, Q_i)$ denotes the set of "threads for which the responses have been returned". We also say Γ *allows*

ℓ, written $\Gamma \vdash \ell$, when $\Gamma \xrightarrow{\ell} \Gamma'$ for some Γ'. Further, $\mathtt{allowed}(\Gamma)$ denotes the set of subjects of the transitions allowed by Γ.

The following notion can be understood as follows: a channel g is pending at some point (corresponding to some process occurrence $\Gamma_i \vdash P_i$) in a transition sequence Φ, if it is allowed (by Γ_i) but it has not received an answer yet.

$$\mathtt{pending}(\Phi,\, \Gamma_i \vdash P_i) \quad \stackrel{\text{def}}{=} \quad \mathtt{allowed}(\Gamma_i) \setminus \mathtt{ended}(\Phi, P_i) \tag{3.1}$$

We call $\mathtt{pending}(\Phi,\, \Gamma_i \vdash P_i)$, "the pending requests at $\Gamma_i \vdash P_i$ in Φ". A pending output is *blocked* if a process can never emit it.

Definition 3.3 (blocked output). Let $\Phi : \Gamma_0 \vdash P_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_i} \Gamma_i \vdash P_i \xrightarrow{l_{i+1}} \cdots$ be a possibly failing transition sequence.

1. c is *blocked at* $\Gamma_i \vdash P_i$ *in* Φ if $c \in \mathtt{pending}(\Phi,\, \Gamma_i \vdash P_i)$ and, moreover, no output at c appears in any possible transition sequence from $\Gamma_i \vdash P_i$ (not restricted to the remaining sequence in Φ).
2. $\mathtt{blocked}(\Phi, \Gamma_i \vdash P_i)$ denotes the set of blocked names in P_i. We sometimes write $\mathtt{blocked}(\Gamma_i \vdash P_i)$ or even $\mathtt{blocked}(P_i)$ when Φ and the typing are clear from the context. We further set $\mathtt{blocked}(\Phi) = \cup_{i \geq 0} \mathtt{blocked}(P_i)$.

Example 3.4 (blocked output). We show how a failing reduction induces a blocked output. Let

$$\mathsf{Lck}\langle u \rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(!u(z).\overline{m}(c)c(h).(\overline{z}|\overline{h}) \,|\, \mathsf{Mtx}\langle m \rangle)$$

This process is a server offering a single operation, which takes a lock and releases immediately. Now consider:

$$\Gamma \vdash \mathsf{Lck}\langle u \rangle \,|\, \overline{u}\langle z' \rangle \,|\, \overline{u}\langle z'' \rangle$$

for a given Γ such that $\Gamma \vdash \overline{z'}$ and $\Gamma \vdash \overline{z''}$. This can reduce to:

$$(\boldsymbol{\nu}\, m)(!u(z).\overline{m}(c)c(h).(\overline{z}|\overline{h}) \,|\, \overline{z'}|(\boldsymbol{\nu}\, h')(\overline{h'} \,|\, h'.\mathsf{Mtx}\langle m \rangle)) \,|\, \overline{u}\langle z'' \rangle;$$

We can now perform a failing reduction at $\overline{h'}$. Then z'' becomes blocked, as m becomes permanently unavailable.

3.2 Intensional Global Progress Properties

We first introduce the notion of resilience which says that some or all threads can potentially survive one or more failures by other threads, i.e. not all threads get blocked by the stalling of one or more other threads. We capture this notion both extensionally and intensionally, then on the basis of these two different characterisations we define intensional and extensional progress properties, respectively. The former faithfully formalise an algorithmic understanding of non-blockingness and wait-freedom, while the latter give their extensional generalisations. We start from the intensional variants. Below, $|S|$ indicates the cardinality of a set S.

Definition 3.5 (strict resilience). A closed process P is *strictly intensionally resilient* or often simply *strictly resilient* if for each finitely failing and fair Φ from P, $|\mathtt{blocked}(\Phi)|$ is no more than the number of failures in Φ.

By bounding the number of blocked outputs to the number of failures, strict resilience ensures that each failure blocks only the component in which it occurs. This definition is intensional because it assumes that the exact number of failures in Φ is known[2]. In § 7, Proposition 6.4, we show that strict resilience is essentially equivalent to *obstruction-freedom* [12].

An interesting aspect of strict resilience is that we can easily weaken the notion by asking that the number of blocked outputs is less than, say, n times the number of failures. We thus obtain a whole range of resilience properties, among which the extensional resilience we shall define later represents a limiting point, while the others are intensional in nature. Examples of (non-)strictly resilient processes follow.

Example 3.6 (strict resilience).

1. Consider the process $\Gamma \vdash \mathsf{Lck}\langle u\rangle \mid \overline{u}\langle z'\rangle \mid \overline{u}\langle z''\rangle \mid \overline{u}\langle z'''\rangle$, which is obtained from Example 3.4, by adding a further request in parallel. After the same reductions as in Ex. 3.4, both z'' and z''' become blocked. Hence the process is not strictly resilient.
2. In contrast, $\mathsf{Ref}\langle u, v\rangle$ (an atomic reference) is strictly resilient, since u is continuously available, it is never possible (due to linearity) that an operation gets blocked.

Strict resilience is the base requirement for a precise formalisation of the current intensional notions of non-blockingness and wait-freedom.

Definition 3.7. $\mathsf{FT}(P)$ denotes the set of fair transition sequences from P containing at most finite failures.

Definition 3.8 (intensional non-blockingness/wait-freedom). A strictly resilient P is:

1. *intensionally non-blocking* (INB) if for any $\Phi \in \mathsf{FT}(P)$ s.t. $\Delta \vdash Q$ is in Φ and $\mathtt{allowed}(\Delta) \setminus \mathtt{blocked}(\Phi) \neq \emptyset$, some output occurs in Φ after Q.
2. *intensionally wait-free* (IWF) if for any $\Phi \in \mathsf{FT}(P)$ s.t. $\Delta \vdash Q$ is in Φ and $c \in \mathtt{allowed}(\Delta) \setminus \mathtt{blocked}(\Phi)$, an output at c occurs in Φ after Q.

In addition to resilience, intensional non-blockingness asks that, in *every* execution, *some* non-blocked outputs eventually come out; wait-freedom replaces "some" with "all". Without resilience, the set $\mathtt{allowed}(\Delta) \setminus \mathtt{blocked}(\Phi)$ could be empty for all Φ and Δ, so that both properties would be trivially satisfied. This is the case when a failure in one component blocks all other components, i.e. any lock-based implementation would become non-blocking, defying our intention. Unlike previous definitions of non-blockingness and wait-freedom, the use of resilience leads to a flexible articulation that can uniformly capture both intensional and extensional properties.

The process $\mathsf{Ref}^{\mathsf{cas}}\langle u, v\rangle$ is a simple example of both intensional non-blockingness and intensional wait-freedom. Further, more complex examples are given in § 4.

3.3 Extensional Global Progress and Inclusion Results

Next we introduce the extensional variant of resilience. The induced global progress properties are strictly more inclusive than their respective intensional counterparts. We then explore the relationships among a variety of extensional progress properties.

[2] Definition 3.5 allows $|\mathtt{blocked}(\Phi)|$ to be less than the number of failures: this is because we wish to include the cases when different requests carry overlapping response channels. In many presentation of algorithms, one restricts the number of threads to execute an operation to one, but in actual implementations, we can easily imagine other possibilities.

Extensional Global Progress. The intensional definitions accurately formalise previous informal presentations. But because of their intensionality, they may not be suitable for compositional reasoning. We now turn our eyes to the extensional global properties, simply by switching from strict resilience to its "observational" counterpart.

Definition 3.9 (extensional resilience, non-blocking, wait-free). P is *extensionally resilient* or simply *resilient* when, for all finitely failing Φ from P, $\mathtt{blocked}(\Phi)$ is a finite set. A process is *non-blocking, NB* (resp. *wait-free, WF*) if it satisfies (1) (resp. (2)) of Def. 3.8, replacing strict resilience with resilience. Henceforth **NB** and **WF** denote, resp., the sets of NB and WF processes.

Extensional resilience abstracts away from counting failures, only requiring that each failure blocks at most finitely many other operations. This abstraction makes sense from an observational viewpoint, just as it makes sense to abstract away from finite τ-actions in weak process equivalences such as weak trace equivalence and weak bisimilarities.

Example 3.10 (extensional progress properties). Suppose a server has two cores, one fragile (a single thread's failure can kill all threads running there) and the other robust. Then a single thread's failure in the former would block all the clients that are using that core, but the remaining clients will continue to work and *will return results* (i.e. are not blocked). For an external observer, there is no difference between this example and all clients in that port failing: observationally, what matters is whether a failure has sufficiently local effects. To abstract away all such local effects, we only demand, under infinite requests, a finite number of failures only block a finite number of threads, i.e. the surviving (non-blocked) threads are co-finite.

Below we write **INB** and **IWF** for the set of processes which are INB (intensionally non-blocking) and IWF (intensionally wait-free), respectively.

Proposition 3.11. *,* $\mathbf{INB} \subsetneq \mathbf{IWF}$*,* $\mathbf{INB} \subsetneq \mathbf{NB}$ *and* $\mathbf{IWF} \subsetneq \mathbf{WF}$.

Proof. In the first clause, the inclusion is immediate from the definition while $\mathsf{CQemp}(r)$ in Section 4 is a differentiating process. The remaining strict inclusions are also immediate (for strictness we can use resources some of whose threads share their fate). □

Relating Extensional Progress Properties. We also define the variants of NB and WF obtained by disabling failures (then by fairness, for any output or conditional that is cyclically enabled, the corresponding reduction or output transition eventually has to occur).

Definition 3.12 (WNB, WWF, RBL). P is *weakly non-blocking* (WNB) (respectively *weakly wait-free*, WWF) if it satisfies (1) (respectively (2)) of Definition 3.8 restricted to non-failing transitions. It is *reliable* (RBL) if it satisfies Definition 3.5 (strict resilience) restricted to non-failing transitions (i.e. there are no blocked outputs at all). We let **RBL / WNB / WWF** denote the sets of RBL/WNB/WWF processes, respectively.

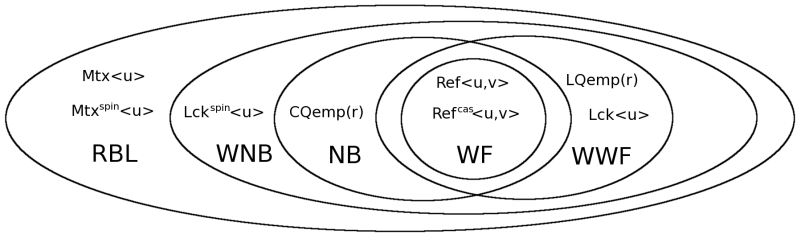
Example 3.13. Two examples:

1. $\mathsf{Lck}\langle u\rangle$ in Example 3.4 is WWF, because by fairness every request is served.
2. Let $\mathsf{Lck}^{\mathsf{spin}}\langle u\rangle$ be the same agent but replacing $\mathsf{Mtx}\langle m\rangle$ with $\mathsf{Mtx}^{\mathsf{spin}}\langle m\rangle$. $\mathsf{Lck}^{\mathsf{spin}}\langle u\rangle$ is WNB but it is not WWF, since in an infinite execution some requests may spin forever.

Proposition 3.14 (relating NB, WF, WNB, WWF, RBL).

1. *$P \approx Q$ implies ($P \in$* **RBL** *$\iff Q \in$* **RBL***).*
2. **NB** $\cup$ **WWF** $\subsetneq$ **WNB** $\subsetneq$ **RBL** *and* **WF** $\subsetneq$ **NB** $\cap$ **WWF**.

(1) is because reliability is an existential requirement. (2) underpins the inclusions among behavioural properties. **NB** $\subsetneq$ **WNB** also says that, if no output comes out in a transition sequence of a non-blocking process, it is due to a failure, not to a functional flaw. The following diagram contains the examples seen so far and two others ($\mathsf{LQemp}(r)$ and $\mathsf{CQemp}(r)$, defined in § 4). Note also that resilience contains NB (by definition) but is incomparable with WNB, WWF and RBL. Finally, as noted in Proposition intensionalextensional, while not in this picture, the intensional versions of **NB** and **WF** are strictly contained in **WF** and **NB** respectively and share $\mathsf{CQemp}(r)$ as a differentiating instance.



4 Application: Semantic Separation of Queues (1)

We now apply our observational theory to the semantic analysis of two imperative queues, one based on a lock mechanism and the other based on the cas operation. The novelty lies in our use of linear actions for rigorous and concise linearisability arguments; and the establishment of an extensional separation.

We first provide an abstract specification of a queue and define the two queues in our calculus (§ 4.1). Correctness is obtained by showing that both queues are bisimilar to the abstract specification.

4.1 Specification

The specification should be as minimal and intuitive as possible, hence we use a state abstraction rather than the π-calculus, as the latter is better suited for an in-depth analysis which is not needed at this point. We first define a *queue state*, denoted $\mathtt{st}, \mathtt{st}', \ldots$, as a triple $\langle \mathsf{Rs},\ \mathsf{Vs},\ \mathsf{As} \rangle$, where (1) Rs is a set of *requests*, each of the form $\mathtt{enq}(v, g)$ or $\mathtt{deq}(g)$ s.t. v and g are respectively its *value* and *continuation name* (2) Vs is a *value sequence* $v_1 \cdots v_n$, s.t. v_1 is the head and v_n is the tail; (3) As is a set of *answers* of the form $\overline{g}\langle \vec{v} \rangle$, in which g is the *continuation name* and $\vec{v}$ a single *value* or ε.

An *abstract queue* $(p, q, \ldots)$ is a pair $\mathsf{AQ}(r, \mathtt{st})$ of a queue state $\mathtt{st}$ and a channel r, known as its *subject* (e.g. $\mathsf{AQ}(r, \langle \{\mathtt{enq}(6, g_1), \mathtt{deq}(g_2)\}, 2{\cdot}3{\cdot}1, \{\overline{g_3}\langle 5 \rangle\} \rangle)$ is an abstract queue with subject r, two requests, three values and one answer). We set

$$\mathsf{AQemp}(r) \stackrel{\text{def}}{=} \mathsf{AQ}(r, \langle \emptyset,\ \emptyset,\ \emptyset \rangle)$$

which denotes the empty queue with subject r.

We define a LTS over abstract queues, where an input corresponds to asynchronously receiving a request and an output corresponds to asynchronously emitting an answer:

$$\begin{array}{rcl}
\mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{r\&\mathtt{enq}\langle v,g\rangle} & \mathsf{AQ}(r,\langle \mathsf{Rs}\cup \mathtt{enq}(v,g),\,\mathsf{Vs},\,\mathsf{As}\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{r\&\mathtt{deq}\langle g\rangle} & \mathsf{AQ}(r,\langle \mathsf{Rs}\cup \mathtt{deq}(g),\,\mathsf{Vs},\,\mathsf{As}\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{(\boldsymbol{\nu}\ g)r\&\mathtt{enq}\langle v,g\rangle} & \mathsf{AQ}(r,\langle \mathsf{Rs}\uplus \mathtt{enq}(v,g),\,\mathsf{Vs},\,\mathsf{As}\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{(\boldsymbol{\nu}\ g)r\&\mathtt{deq}\langle g\rangle} & \mathsf{AQ}(r,\langle \mathsf{Rs}\uplus \mathtt{deq}(g),\,\mathsf{Vs},\,\mathsf{As}\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs}\uplus \mathtt{enq}(v,g),\,\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{\tau} & \mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs}\cdot v,\,\mathsf{As}\uplus \overline{g}\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs}\uplus \mathtt{deq}(g),\,v\cdot\mathsf{Vs},\,\mathsf{As}\rangle) & \xrightarrow{\tau} & \mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\uplus \overline{g}\langle v\rangle\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs}\uplus \mathtt{deq}(g),\,,\,\mathsf{As}\rangle) & \xrightarrow{\tau} & \mathsf{AQ}(r,\langle \mathsf{Rs},\,,\,\mathsf{As}\uplus \overline{g}\langle KO\rangle\rangle)\\
\mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\uplus \overline{g}\langle v\rangle\rangle) & \xrightarrow{\overline{g}\langle v\rangle} & \mathsf{AQ}(r,\langle \mathsf{Rs},\,\mathsf{Vs},\,\mathsf{As}\rangle)
\end{array}$$

Above, $\mathsf{Rs}\uplus\mathtt{deq}(g)$ is the union of Rs and $\{\mathtt{deq}(g)\}$ such that $\mathtt{deq}(g)\notin\mathsf{Rs}$. All τ-actions represent state changes, which we call *commit actions* or more simply *commits*. We write them as $\mathsf{AQ}(r,\mathtt{st})\xrightarrow{\mathbf{com}(g)}\mathsf{AQ}(r,\mathtt{st}')$.

Convention 4.1 (distinct continuations). *Henceforth, we assume that the continuation names received in the input transitions are always bound, i.e. we assume that the first two cases of transitions above never occur. Thus, starting from the empty queue, all continuations inside an abstract queue are distinct. This assumption does not make us lose generality because all the arguments in the correctness proofs apply just as well if we index each continuation name with a fresh index.*

To define a consistent typed LTS over abstract queues, we need to equip abstract queues with linear typing, which is easily done. For example, the queue above is typed under $r:\&^{\star}\{\mathtt{enq}(\mathtt{int}\uparrow^{\star}()),\mathtt{deq}(\uparrow^{\star}(\mathtt{int})\},\ g_1:\uparrow^{\star}(),\ g_2:\uparrow^{\star}(\mathtt{int}),\ g_3:\uparrow^{\star}(\mathtt{int})$.

4.2 Two Queues

Lock-based Queue. Next we introduce a process encoding of a lock-based queue, which implements the specification using a lock.

$$\begin{array}{rcl}
\mathsf{LQ}(r,h,t,l) & \stackrel{\text{def}}{=} & !\,r\&\{enq(v,u):\bar{l}(g)g(y).P^{lck}_{enq}(v,t,y,u),\ deq(u):\bar{l}(g)g(y).P^{lck}_{deq}(h,t,y,u)\}\\
\mathsf{LQemp}(r) & \stackrel{\text{def}}{=} & (\nu h,t,s,l)(\mathsf{Mtx}\langle l\rangle|\mathsf{LQ}(r,h,t,l)|\mathsf{LPtr}(h,s)|\mathsf{LPtr}(t,s)|\mathsf{LENode}(s,0))
\end{array}$$

where $\mathsf{LQemp}(r)$ is the empty configuration and $\mathsf{LPtr}(h,s)$, $\mathsf{LPtr}(t,s)$ and $\mathsf{LENode}(s,0)$ are (non-CAS) references from § 2. The queue is represented as a linked list. Pointers h and t store the names of the head and tail nodes, respectively: when they coincide, the list is empty, with a single dummy node (as above). The key steps are the *non-linear* interactions with $\mathsf{Mtx}\langle l\rangle$. The remaining behaviours $P^{lck}_{enq}(v,t,y)$ and $P^{lck}_{deq}(h,t,y)$ are the obvious list manipulation followed by lock relinquishment (see § D).

CAS-based Queue. The last is a cas-based non-blocking queue due to [22]:

$$\begin{array}{rcl}
\mathsf{CQ}(r,h,t) & \stackrel{\text{def}}{=} & !\,r\&\{\mathtt{enq}(v,g):\mathsf{Enqueue}_{\text{cas}}\langle t,v,g\rangle,\ \ \mathtt{deq}(g):\mathsf{Dequeue}_{\text{cas}}\langle h,t,g\rangle\}\\
\mathsf{CQemp}(r) & \stackrel{\text{def}}{=} & (\nu h,t,nd_0,nxt_0)(\mathsf{CQ}(r,h,t)\mid\mathsf{Ptr}(h,nd_0,0)\mid\mathsf{Ptr}(t,nd_0,0)\\
 & & \qquad\mid\mathsf{Node}(nd_0,0,nxt_0)\mid\mathsf{Ptr}(nxt_0,\mathsf{null},0))
\end{array}$$

The cas-based queue is also represented as a linked list. A *node* $\mathsf{Node}(nd, v, ptr)$ is a reference $\mathsf{Ref}\langle nd, \langle v, ptr\rangle\rangle$ storing a value and the name of a *pointer* $\mathsf{Ptr}(ptr, nxt, ctr)$, which is a cas-reference $\mathsf{Ref}^{\mathsf{cas}}\langle ptr, \langle nxt, ctr\rangle\rangle$, containing the name of the next node, or null, and a *counter* incremented at each successful cas. To scan, we start from h, reach the initial (dummy) node, get the pointer name and reach the first value node, and so on. The enqueue operation $\mathsf{Enqueue}_{\mathsf{cas}}\langle x, tail, u\rangle$ is the key algorithm:

```
1  Enqueue_cas⟨x, tail, u⟩ =
2    (ν node)((ν nlPtr)(Ptr(nlPtr, null, 0) | Node(node, x, nlPtr)) |
3    (μX_tag(u′).
4       tail ◁ read(last, ctrT).
5       last ◁ read(tPtr, ∗).
6       tPtr ◁ read(next, ctr).
7       if^L (next = null) then
8         if^L cas(tPtr, ⟨next, ctr⟩, ⟨node, ctr + 1⟩) then
9           CAS(t, ⟨last, ctrT⟩, ⟨node, ctrT + 1⟩);
10          u′
11        else X_tag⟨u′⟩
12      else CAS(t, ⟨last, ctrT⟩, ⟨next, ctrT + 1⟩); X_tag⟨u′⟩
13   )⟨u⟩)
```

The notations $\texttt{if cas}(u, v, w)\ \texttt{then}\ P\ \texttt{else}\ Q$ and $\texttt{CAS}(u, v, w); P$ are from § 2; the notation $x \triangleleft read(\vec{y}).P$ is short for $(\nu c)(\overline{x} \oplus read\langle c\rangle | c^L(\vec{y}).P)$, where $*$ is for irrelevant values. $\mathsf{Enqueue}_{\mathsf{cas}}\langle x, t, u\rangle$ uses cas to *append* a node and to *swing* the tail pointer t. The dequeue operation $\mathsf{Dequeue}_{\mathsf{cas}}\langle h, t, g\rangle$ is defined as follows:

```
1  Dequeue_cas⟨head, tail, u⟩ = (μX_tag(u′).
2       head ◁ read(hn, h_ctr).
3       tail ◁ read(tn, t_ctr).
4       hn ◁ read(∗, hp).
5       hp ◁ read(next, ∗).
6       if^L (hn = tn) then
7         if^L (next = null) then
8           u′⟨null⟩
9         else
10          CAS(tail, ⟨tn, t_ctr⟩, ⟨next, t_ctr + 1⟩); X_tag⟨u′⟩
11      else
12        next ◁ read(x, ∗).
13        if^L (cas(head, ⟨hn, ctr⟩, ⟨next, ctr + 1⟩)) then
14          u′⟨x⟩
15        else X_tag⟨u′⟩)⟨u⟩
```

Above, $\mathsf{Dequeue}_{\mathsf{cas}}\langle head, tail, u\rangle$ and $\mathsf{Enqueue}_{\mathsf{cas}}\langle x, tail, u\rangle$ are slightly simplified versions of the corresponding algorithms in [22] (the latter can be obtained by adding a few more checks for run-time optimisations). $\mathsf{CQemp}(r)$, $\mathsf{LQemp}(r)$ and $\mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle)$ are all typed under $r : \&^\star\{\texttt{enq}(\alpha \uparrow^\star ()), \texttt{deq}(\uparrow^\star (\alpha)\}$, for some type α.

4.3 Functional Correctness

cas-based Queues. We now establish the functional correctness of $\mathsf{CQemp}(r)$ by showing that it is weakly bisimilar to $\mathsf{AQemp}(r)$, the empty abstract queue. We give the outline of key arguments, leaving details to § G.

Henceforth we reason using *molecular actions*, denoted by $P \stackrel{\ell}{\mapsto} Q$ and consisting of a transition $P \stackrel{\ell}{\rightarrow}$, followed by all available linear actions. This definition is justified by Prop. 2.3(1,2). Molecular actions give high-level abstraction in operational proofs without changing the nature of the semantic notions, e.g. the use of $\stackrel{\ell}{\mapsto}$ as one-step transition does not alter the notion of $\approx$, nor the global progress properties. We call cas-*queue process* any molecular action derivative of $\mathsf{CQemp}(r)$.

We narrow down the length of the bisimilarity proof $\mathsf{CQemp}(r) \approx \mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset \rangle)$ by showing that any cas-queue process can be reduced to a unique normal form through a sequence of internal non-commit actions (both of these notions are defined below), and then reasoning on normal forms only. But before we do that, we need to define the general form of such processes, as follows.

Definition 4.2 (general form). A cas-queue process P is in *general form* when:

$$P \;\equiv\; (\nu h, t, nd_0..nd_n)(\; \mathsf{CQ}(r, h, t) \;\mid\; \textstyle\prod_{1 \leq i \leq m} P_i \;\mid\; LL \;)$$

typed under

$$r : \&\{\mathtt{enq}(\alpha \uparrow ()), \mathtt{deq}(\uparrow (\alpha)\},\ \{g_i :\uparrow (\vec{\alpha})\}_{1 \leq i \leq n},$$

for some α; where each P_i is in local molecular form (as we shall define shortly) and contains a single free occurrence of the name g_i (we say P_i is a g_i*-thread*, or simply *thread*, *of* P); and LL ("*linked list sub-process*") has the following form:

$$\mathsf{Ptr}(h, \langle nd_H, ctr_H \rangle) \mid \mathsf{Ptr}(t, \langle nd_T, ctr_T \rangle) \mid \textstyle\prod_{1 \leq i \leq n} \mathsf{NnP}\langle nd_i, v_i, nd_{i+1}, ctr_i \rangle$$

where we set $\mathsf{NnP}\langle nd, v, nd', ctr \rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, nxt)(\mathsf{Node}(nd, v, nxt) | \mathsf{Ptr}(nxt, nd', ctr))$ such that $nd_{n+1} = null$, $0 \leq H \leq T \leq n$ and either $T = n - 1$, then we say LL is *pre-quiescent*; or $T = n$, then we say LL is *quiescent*. In both, nd_{T+1} is called the *successor of the tail*, h the *head pointer*, and t the *tail pointer*.

We observe:

- nd_H is the head of the linked list LL (the nodes from nd_0 to nd_{H-1} have already been dequeued and are semantic garbage).
- nd_T may be the last or the second to last node in LL, for $n - 1 \leq T \leq n$ (however, t always points to the last node when the queue is in *normal form*, defined later).
- LL is quiescent when the successor of the tail is $null$. By construction, if a pointer in LL has a $null$ successor field then it is the final node in the linked-list.

Convention 4.3 (distinct continuations). *Above and henceforth, we use continuation names to index threads, assuming that all continuation names a process receives in requests are fresh names, i.e. assuming all inputs are bound inputs w.r.t. received names. This assumption reflects Convention 4.1. It does not lose generality at the level of bisimilarity in the absence of name matching (name comparison) [24, 33]. All arguments including bisimilarity can also be carried out by indexing processes following e.g. [4] rather than using name indexes.*

We now define the local molecular forms that we mentioned earlier, which give the syntactic shapes of cas-queue processes modulo linear reductions. Below the notation

$\mathsf{Enq}^{(n)}_{\mathsf{cas}}\langle v_1,\ldots,v_m\rangle$ denotes the sub-process of the enqueue process $\mathsf{Enqueue}_{\mathsf{cas}}\langle t,v,g\rangle$, up to arbitrarily many unfoldings of recursion, such that (1) it starts from Line n; (2) it does not contain a free recursion variable; and (3) v_1,...,v_m are instantiated in its free value variables in this order (omitting the substitutions that are not used anymore). Similarly we define $\mathsf{Deq}^{(n)}_{\mathsf{cas}}\langle v_1,\ldots,v_m\rangle$.

Definition 4.4 (local molecular form, LMF). A *local molecular form*, or *LMF* henceforth, *for an enqueue operation*, is one of the following processes. Below in the 2nd-5th lines, we set $C[\,\cdot\,]$ as $(\boldsymbol{\nu}\, nd)([\,\cdot\,]\,|\,(\boldsymbol{\nu}\, m)(\mathsf{Node}(nd,v,m)|\mathsf{Ptr}(m,\mathsf{null},0)))$.

$$\begin{aligned}
\mathsf{EnqReq}_{\mathsf{cas}}\langle r,v,g\rangle &\stackrel{\text{def}}{=} \overline{r}\oplus \mathtt{enq}\langle v,g\rangle\\
\mathsf{EnqRdT}_{\mathsf{cas}}\langle t,v,g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(4)}_{\mathsf{cas}}\langle t,nd,g\rangle]\\
\mathsf{EnqRdTN}_{\mathsf{cas}}\langle tn,t_ctr,t,v,g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(5)}_{\mathsf{cas}}\langle tn,t_ctr,t,nd,g\rangle]\\
\mathsf{EnqRdTP}_{\mathsf{cas}}\langle tp,tn,t_ctr,t,v,g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(6)}_{\mathsf{cas}}\langle tp,tn,t_ctr,t,nd,g\rangle]\\
\mathsf{EnqCom}_{\mathsf{cas}}\langle nxt,tp_ctr,tp,tn,t_ctr,t,v,g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(8)}_{\mathsf{cas}}\langle nxt,tp_ctr,t_ctr,tn,t,nd,g\rangle]\\
\mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn,t_ctr,t,nd,g\rangle &\stackrel{\text{def}}{=} \mathsf{Enq}^{(9)}_{\mathsf{cas}}\langle tn,t_ctr,t,nd,g\rangle\\
\mathsf{EnqAns}_{\mathsf{cas}}\langle g\rangle &\stackrel{\text{def}}{=} \mathsf{Enq}^{(10)}_{\mathsf{cas}}\langle g\rangle \stackrel{\text{def}}{=} \overline{g}\\
\mathsf{EnqSwRec}_{\mathsf{cas}}\langle tn,t_ctr,t,nd,g\rangle &\stackrel{\text{def}}{=} \mathsf{Enq}^{(12)}_{\mathsf{cas}}\langle tn,t_ctr,t,nd,g\rangle
\end{aligned}$$

Similarly a *LMF for a dequeue operation* is as one of the following processes.

$$\begin{aligned}
\mathsf{DeqReq}_{\mathsf{cas}}\langle r,g\rangle &\stackrel{\text{def}}{=} \overline{r}\oplus \mathtt{deq}\langle g\rangle\\
\mathsf{DeqRdH}_{\mathsf{cas}}\langle h,t,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(2)}_{\mathsf{cas}}\langle h,t,g\rangle\\
\mathsf{DeqRdT}_{\mathsf{cas}}\langle hn,h_ctr,h,t,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(3)}_{\mathsf{cas}}\langle hn,h_ctr,h,t,g\rangle\\
\mathsf{DeqRdHN}_{\mathsf{cas}}\langle tn,t_ctr,hn,h_ctr,h,t,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(4)}_{\mathsf{cas}}\langle tn,t_ctr,hn,h_ctr,h,t,g\rangle\\
\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt,hp_ctr,tn,t_ctr,&\\
hn,h_ctr,h,t,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(5)}_{\mathsf{cas}}\langle nxt,tp_ctr,tn,t_ctr,hn,h_ctr,h,t,g\rangle\\
\mathsf{DeqAnsNull}_{\mathsf{cas}}\langle g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(8)}_{\mathsf{cas}}\langle g\rangle \stackrel{\text{def}}{=} \overline{g}\langle null\rangle\\
\mathsf{DeqSw}_{\mathsf{cas}}\langle nxt,t_ctr,tn,h,t,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(10)}_{\mathsf{cas}}\langle nxt,t_ctr,tn,h,t,g\rangle\\
\mathsf{DeqRdNext}_{\mathsf{cas}}\langle h_ctr,hn,h,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(12)}_{\mathsf{cas}}\langle t,tn,h_ctr,hn,h,g\rangle\\
\mathsf{DeqCom}_{\mathsf{cas}}\langle x,t,h_ctr,hn,h,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(13)}_{\mathsf{cas}}\langle x,t,h_ctr,hn,h,g\rangle\\
\mathsf{DeqAns}_{\mathsf{cas}}\langle v,g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(14)}_{\mathsf{cas}}\langle v,g\rangle \stackrel{\text{def}}{=} \overline{g}\langle v\rangle
\end{aligned}$$

A LMF for either an enqueue operation or a dequeue operation, is simply called *LMF*. In a LMF, g in each line is called the *continuation name* of the process.

Intuitively, the evolution of an enqueue LMF within a cas-queue process follows the pattern given in Fig. 1, while a dequeue LMF follows Fig. 2. Labels on arrows indicate commit and swinging actions: a *commit action*, denoted $P \stackrel{\mathsf{com}(g)}{\longmapsto} Q$, is a molecular action which makes an irreversible state change on the linked list; a *swinging action*, denoted $P \stackrel{\mathsf{sw}(g)}{\longmapsto} Q$, is a molecular action which corresponds to a cas operation on the

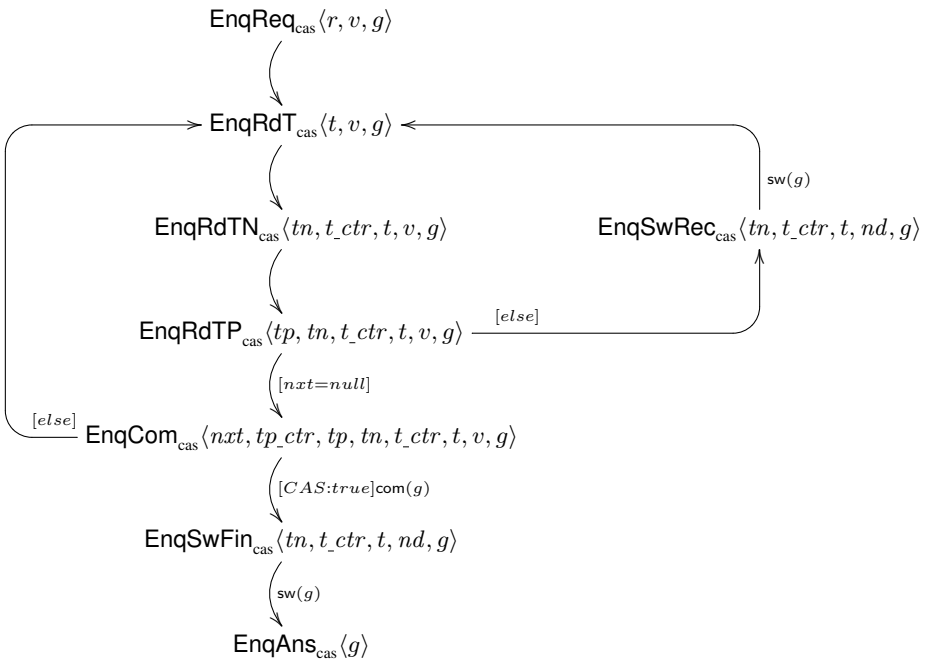


Fig. 1. Evolution patterns of LMFs for enqueue operations.

tail pointer to swing it forward to the next element in the linked list; we also define a *non-commit action*, denoted $P \xmapsto{\mathsf{nc}(g)} Q$, as any internal molecular action which is not a commit. In Fig. 1 and 2 we have omitted non-commit actions since they can be easily inferred (note that swinging actions are also non-commit). Labels in Fig. 1 and 2 also indicate the outcome of the evaluation of a condition (within square brackets), where the given molecular transition includes the reduction of a (linear) conditional. Note that an action from some LMF may be a commit or not, according to the outcome of a condition which is evaluated within the action itself.

With LMFs we have completed the definition of general form. Next, we show that this is indeed a general form (i.e. that it is satisfied by any cas-queue process) as well as two additional key invariants. The following definition is required.

Definition 4.5 (alignment). Let P be a cas-queue process and let P_i be a thread of P that has read the contents of a pointer x in the linked-list sub-process of P. Let ctr_x be the value of the counter field that was read from x. We say P_i is *aligned* (resp. *pre-aligned*) at x when ctr_x is equal to (resp. less than) the current value in the counter field of x.

We can now state the key invariants of cas-queue processes.

Proposition 4.6 (invariants in cas-queue processes). *Let P be a* cas*-queue process.*

(a) *P is in general form.*
(b) *The linked-list sub-process is quiescent if the successor of the tail is $null$, otherwise it is pre-quiescent.*
(c) *Each thread P_i of P that has read the contents of a pointer x, is either aligned or pre-aligned at x.*

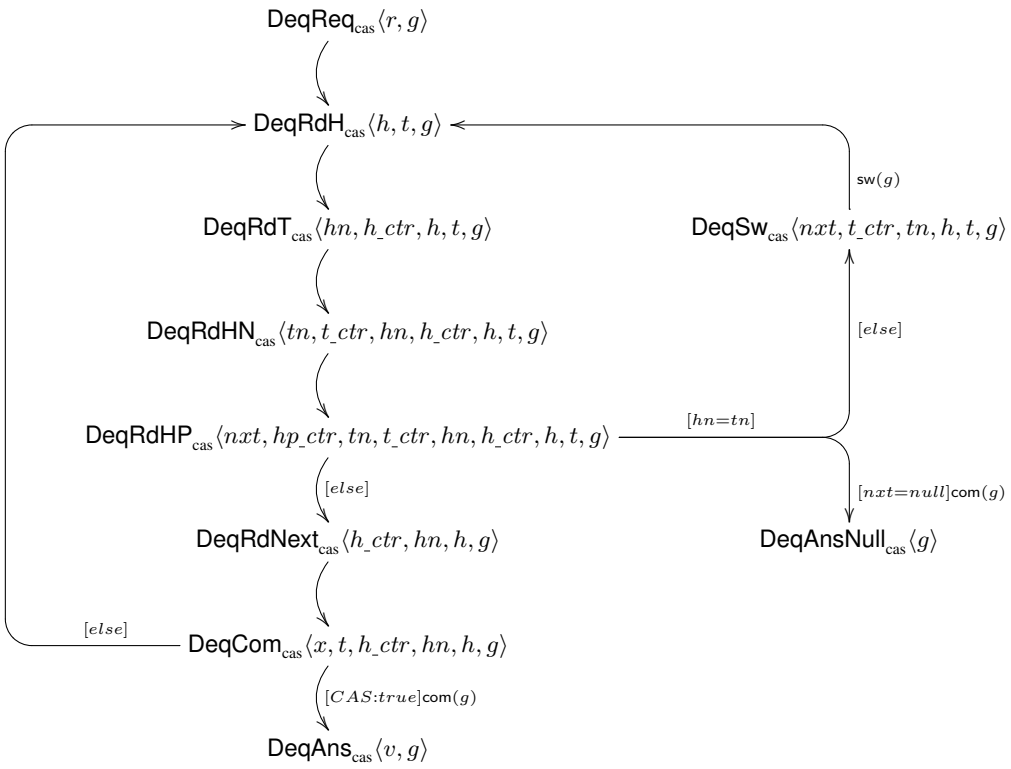


Fig. 2. Evolution patterns of LMFs for dequeue operations.

Proof. Suppose $\mathsf{CQemp}(r) \stackrel{\ell_1}{\longmapsto} .. \stackrel{\ell_n}{\longmapsto} P$. We establish (a), (b) and (c) simultaneously, by induction on n.
(Base Step.) This is when $n = 0$ and $P = \mathsf{CQemp}(r)$.

(a) Immediate since the linked list has only the dummy node and there are no threads.
(b) The linked-list is quiescent and the successor of the node recovered from t is $null$.
(c) Vacuous since no thread exists.

(Inductive Step.) We assume the result holds for $n = m$, and show it for $n = m+1$. Let:

$$\mathsf{CQemp}(r) \stackrel{\ell_1}{\longmapsto} .. \stackrel{\ell_m}{\longmapsto} P \stackrel{\ell_{m+1}}{\longmapsto} P'.$$

By induction hypothesis, P is in general form, so that each P_i-component contains a continuation name and has a prefix which is either part of a redex or an output of an answer. By the typing, ℓ_{m+1} can be either the input of a request, the output of an answer, or a τ-action. The first two cases are almost trivial (Proposition G.11 (2)). If $\ell_{m+1} = \boldsymbol{\tau}$, it can be shown that the prefix of one of the threads, say P_i, is reduced by ℓ_{m+1} (Proposition G.11 (1)). Then we reason on the shape of P_i.

Case $\mathsf{EnqReq}_{\mathbf{cas}}\langle v, g\rangle$ **(request):** Then the involved reduction is:

$$\mathsf{EnqReq}_{cas}\langle r, v, g\rangle|\mathsf{CQ}(r, h, t) \stackrel{\tau}{\longmapsto} \mathsf{EnqRdT}_{cas}\langle t, v, g\rangle|\mathsf{CQ}(r, h, t)$$

reducing to a LMF without changing the contents of h and t nor the linked-list. Hence **(a)**, **(b)** and **(c)** immediately hold.

The same reasoning applies to the case of the dequeue request, $\mathsf{DeqReq}_{\mathsf{cas}}\langle r, v, g\rangle$.

Case $\mathsf{EnqRdT}_{\mathbf{cas}}\langle t, v, g\rangle$ (read operation): In this case, P_i reads a pointer:

$$\mathsf{EnqRdT}_{\mathsf{cas}}\langle t, v, g\rangle|\mathsf{Ptr}(t, \langle tn, t\_ctr\rangle) \overset{\tau}{\longmapsto} \mathsf{EnqRdTN}_{\mathsf{cas}}\langle tn, t\_ctr, t, v, g\rangle|\mathsf{Ptr}(t, \langle tn, t\_ctr\rangle)$$

Note that the existence of the pointer t follows from the fact the P is in general form (induction hypothesis). In the result, the process is aligned at t. The remaining conditions are from induction hypothesis since the linked list has not been modified.

The same reasoning applies to all the other read operations (both on nodes and on pointers), which are given by the following cases of LMF:

$$\mathsf{EnqRdTN}_{\mathsf{cas}}\langle tn, t\_ctr, t, v, g\rangle,\ \mathsf{EnqRdTP}_{\mathsf{cas}}\langle tp, tn, t\_ctr, t, v, g\rangle,\ \mathsf{DeqRdH}_{\mathsf{cas}}\langle h, t, g\rangle,$$
$$\mathsf{DeqRdT}_{\mathsf{cas}}\langle hn, h\_ctr, h, t, g\rangle,\ \mathsf{DeqRdHN}_{\mathsf{cas}}\langle tn, t\_ctr, hn, h\_ctr, h, t, g\rangle,$$
$$\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt, hp\_ctr, tn, t\_ctr, hn, h\_ctr, h, t, g\rangle,\ \mathsf{DeqRdNext}_{\mathsf{cas}}\langle h\_ctr, hn, h, g\rangle$$

In particular, the existence of each node/pointer is also ensured by induction hypothesis (i.e. since P is in general form, both the head and the tail pointer point to existing nodes and each node in the linked list refers to an existing pointer).

Case $\mathsf{EnqCom}_{\mathbf{cas}}\langle nxt, tp\_ctr, tp, tn, t\_ctr, t, v, g\rangle$ (cas operation): We know that nxt is $null$ because the current LMF can only be reached after checking this condition. This means that tp had not been modified prior to its reading, then we infer that $tp\_ctr = 0$.

If P_i is aligned at tp then the linked list sub-process must be quiescent, because tp was accessed from t (through tn). The cas-operation makes nxt point to the new node:

$$\mathsf{EnqCom}_{\mathsf{cas}}\langle nxt, tp\_ctr, tp, tn, t\_ctr, t, v, g\rangle|\mathsf{Ptr}(tp, \langle null, 0\rangle) \overset{\tau}{\longmapsto} \mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle|\mathsf{Ptr}(tp, \langle nd, 1\rangle)$$

Note that the remaining pending processes, if any, automatically become pre-aligned as tp_ctr is incremented. Also, since the linked list was quiescent before the transition, it becomes pre-quiescent after the transition. Then (a), (b) and (c) are satisfied.

If P_i is not aligned at tp, then both fields of tp have changed. As a result, no change is made to the linked list and (a), (b) and (c) are again satisfied:

$$\mathsf{EnqCom}_{\mathsf{cas}}\langle nxt, tp\_ctr, tp, tn, t\_ctr, t, v, g\rangle|\mathsf{Ptr}(tp, \langle nxt', tp\_ctr'\rangle) \overset{\tau}{\longmapsto} \mathsf{EnqRdT}_{\mathsf{cas}}\langle t, v, g\rangle|\mathsf{Ptr}(tp, \langle nxt', tp\_ctr'\rangle)$$

A similar reasoning applies to the other cas operations, i.e. from the following LMFs:

$$\mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle,\quad \mathsf{EnqSwRec}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle,$$
$$\mathsf{DeqSw}_{\mathsf{cas}}\langle nxt, t\_ctr, tn, h, t, g\rangle,\quad \mathsf{DeqCom}_{\mathsf{cas}}\langle x, t, h\_ctr, hn, h, g\rangle$$

In particular, the last case is a cas on the head pointer h. In this case quiescence (or pre-quiescence) is preserved, since neither the tail pointer nor the last pointer in the linked list sub-process are modified. Note also that the LMF is reached after checking that the tail and head pointers point to different nodes. Then by induction hypothesis, t must point to a later node (higher index) than h, which in turn means that the successor of the node which was accessed from h is not $null$. Hence after the transition, h would still point to some node in the linked list and all the three invariants are satisfied.

The other cases above are all cas operations on the tail pointer t. In each case, it is checked that the successor of the tail is not $null$ before reaching the LMF. Then a successful cas in this case would make a pre-quiescent linked list into a quiescent one, while satisfying the other invariants as well. □

Normal forms are special cases of general forms. We define local and global normal forms, in this order.

Definition 4.7 (local normal form, LNF). We say that a LMF P_i in a cas-queue process is a *local normal form (LNF)* when one of the following two conditions holds.

1. P_i is ready to commit in the next step, i.e. it has one of the following forms:
   (a) $\mathsf{EnqCom}_{\mathsf{cas}}\langle nxt, tp\_ctr, tp, tn, t\_ctr, t, v, g\rangle$,
   (b) $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt, hp\_ctr, tn, t\_ctr, hn, h\_ctr, h, t, g\rangle$, and
   (c) $\mathsf{DeqCom}_{\mathsf{cas}}\langle x, t, h\_ctr, hn, h, g\rangle$.
   Then we say that P_i is a *pending process*.
2. P_i is an answer to an enqueue/dequeue request, that is when it has one of the following forms, $\mathsf{DeqAns}_{\mathsf{cas}}\langle v, g\rangle$, $\mathsf{DeqAnsNull}_{\mathsf{cas}}\langle g\rangle$, or $\mathsf{EnqAns}_{\mathsf{cas}}\langle g\rangle$. Then we say that P_i is an *answer at g* or simply an *answer*.

Definition 4.8 (normal form, NF). A *normal form* is a quiescent cas-queue process in general form, whose threads are all in LNF.

A critical argument is that from any general form we can reach a normal form by a sequence of non-commit actions. Such a sequence is called normalisation. Formally:

Definition 4.9 (normalisation). Let P be a cas-queue process.

1. Assume that P contains a g-thread. Then we write $P \xmapsto{\mathsf{norm}(g)} P'$ if there is a sequence of zero or more molecular τ-actions from P to P', reducing the g-thread of P, such that the g-thread in P' is in LNF. $\xmapsto{\mathsf{norm}(g)}$ is called a *local normalisation of P at g*, which is *committing* if it contains a commit action; is *post-committing* if the g-thread in P has already committed; and is *pre-committing* if it is neither.
2. Let $g_1, g_2, .., g_n$ be threads of P. Then $P \xmapsto{\mathsf{norm}(g_1 g_2 .. g_n)} P'$ denotes

$$(*) \qquad P \xmapsto{\mathsf{norm}(g_1)} \xmapsto{\mathsf{norm}(g_2)} .. \xmapsto{\mathsf{norm}(g_n)} P'$$

   where, for some j s.t. $1 \leq j \leq n$, each $\xmapsto{\mathsf{norm}(g_i)}$ for $1 \leq i \leq j$ in $(*)$ is committing or post-committing, while the remaining are pre-committing. We call $(*)$ above, a *(global) normalisation of P*.

In (1), if the g-thread is already in normal form, its local normalisation is going to be empty. In (2), the sequence of local normalisations is partitioned into committing/post-committing and pre-committing, because a committing local normalisation *after* a pre-committing one can invalidate the latter's normal form.

We want to show that any cas-queue process P in general form admits a global normalisation linearisable according to an arbitrary partition of its threads. For that purpose, we use the following local permutations.

Proposition 4.10 (local permutation). *Let P be a* cas*-queue process.*

1. (nc-up) *$P \xmapsto{\mathsf{norm}(g_j)} \xmapsto{\mathsf{nc}(g_i)} R$ and $P \xmapsto{\mathsf{nc}(g_i)}$ imply $P \xmapsto{\mathsf{nc}(g_i)} \xmapsto{\mathsf{norm}(g_j)} \xmapsto{\mathsf{nc}(g_i)}{}^{*}$ R' such that $R \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$.*

2. (sw-com) $P \xmapsto{\mathsf{sw}(g_i)} \xmapsto{\mathsf{com}(g_j)} R$ *implies* $P \xmapsto{\mathsf{com}(g_j)} \xmapsto{\mathsf{sw}(g_i)} R$.

The proofs of these results are given in § G.3. For (nc-up), we had to require $P \xmapsto{\mathsf{nc}(g_i)}$ because the read operation from $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt, hp\_ctr, tn, t\_ctr, hn, h\_ctr, h, t, g\rangle$ is usually a non-commit action, but when it is moved left of an enqueue commit it *may* become a commit (i.e. it is a commit only if the queue is empty). Since this particular permutation is not used in the normalisation proof, we rule it out by requiring $P \xmapsto{\mathsf{nc}(g_i)}$.

The following is the anticipated normalisation lemma. Below, a cas-queue process is *initial* if it has the form $\mathsf{CQemp}(r, h, t) \mid \prod_{1\le i\le m} P_i$, where P_i is either of the form $\mathsf{EnqReq}_{\mathsf{cas}}\langle r, v_i, g_i\rangle$ or of the form $\mathsf{DeqReq}_{\mathsf{cas}}\langle r, g_i\rangle$, for all $1 \le i \le m$. Also, $P_0 \xmapsto{\tau}{}^{(n)} Q$ stands for a sequence of n molecular actions $P_0 \xmapsto{\tau}{}^{*} Q$.

Lemma 4.11 (normalisation). *Let P_0 be initial and let $P_0 \xmapsto{\tau}{}^{(n)} Q$ $(n \ge 0)$, where the $K_0 = g_1 \ldots g_i$-threads $(i \ge 0)$ are all and the only threads in which a commit takes place (in this order). Let K be a non-redundant sequence of all the threads in P_0 which can be partitioned as $K_0 \cdot K_1 \cdot K_2$. Then $Q \xmapsto{\mathsf{norm}(K)} R$ and $P_0 \xmapsto{\mathsf{norm}(K)} R$ for some R such that all the threads of the partition $K_0 \cdot K_1$ contain a commit in $P_0 \xmapsto{\mathsf{norm}(K)} R$, while those of the partition K_2 do not.*

Proof. By induction on n in $P_0 \xmapsto{\tau}{}^{(n)} Q$. Below $Q \xmapsto{\mathsf{norm}(K)} R$ in the statement is called *completion* while $P_0 \xmapsto{\mathsf{norm}(K)} R$ is the (corresponding) *linearisation*.

(Base Case) The case of $n = 0$ consists in sequentially normalising the threads of an initial process in an arbitrary order. Since the process is initial, each normalisation only requires one iteration (cf. Figures 1 and 2). The full proof is shown in § G (Prop. G.13).

(Induction Case) Assume the statement holds for $n = m$ and

$$P_0 \xmapsto{\tau}{}^{(m)} Q_0 \xmapsto{\tau} Q \tag{4.1}$$

in which all and only the K_0-threads commit, in the order of K_0. By induction hypothesis, for any K_1 and K_2 which partition the remaining threads, we have:

$$\Phi : Q_0 \xmapsto{\mathsf{norm}(K_0\cdot K_1\cdot K_2)} R \tag{4.2}$$

and

$$\Psi : P_0 \xmapsto{\mathsf{norm}(K_0\cdot K_1\cdot K_2)} R \tag{4.3}$$

where the local normalisations for the $(K_0 \cdot K_1)$-threads are committing in Ψ and those for the K_2-threads are pre-committing. Now let

$$\Phi_0 : Q_0 \xmapsto{\tau} Q \tag{4.4}$$

and assume, w.l.o.g., that the g'-thread is the one involved in this molecular action. Then there are two cases of Φ_0 (below we use some minor results which shall be shown in § G):

1. **(Φ_0 is a non-commit action at g')** We focus on Φ in (4.2). Note that the g'-thread in Q_0 is not a local normal form [because if it was pending then Φ_0 should be a commit

and if it was an answer then Φ_0 should be an output transition]. Hence Φ contains at least one molecular τ-action at g' by Proposition G.7. By taking the first molecular τ-action at g' in Φ and applying Proposition 4.10 (nc-up) repeatedly, we are able to obtain a normalisation sequence after Q:

$$\Phi' : Q_0 \xmapsto{\mathsf{nc}(g')} Q \xmapsto{\mathsf{norm}(K_0 \cdot K_1 \cdot K_2)} R \tag{4.5}$$

It can be shown that the first $\xmapsto{\mathsf{nc}(g')}$-action coincides with Φ_0 (Proposition G.7) and that the remaining normalisations in Φ' give the desired completion from Q. While Ψ is still the desired linearisation.

2. (Φ_0 **is a commit action at** g'**)** Then g' is not in K_0, since a thread may not commit twice (Proposition G.12). Again we focus on Φ in (4.2). Note that, having fixed K_0, the induction hypothesis holds for any partition $K_0 \cdot K_1 \cdot K_2$. Then consider the case where K_1 starts with g', i.e. Φ is equivalently written as follows:

$$\Phi : Q_0 \xmapsto{\mathsf{norm}(K_0)} \xmapsto{\mathsf{norm}(g')} \xmapsto{\mathsf{norm}(K_1' \cdot K_2)} R \tag{4.6}$$

where $K_1 = g' \cdot K_1'$. Since the normalisations in $Q_0 \xmapsto{\mathsf{norm}(K_0)}$ are all post-committing, they may only contain swinging actions (Proposition G.12). Note also that the normalisation sequence for g' above starts with a commit at g' [by contradiction, suppose it starts with a non-commit action. Since $Q_0 \xmapsto{\mathsf{norm}(K_0)}$ only contains tail-swinging operations, it is easy to see that the same non-commit action at g' would be enabled in Q_0 as well. Then g' would be able to perform two different actions in Q_0, but it can be shown that this is not possible (Prop. G.7)]. Then the commit at g' can be permuted up by repeated applications of Proposition 4.10 (sw-com), obtaining:

$$Q_0 \xmapsto{\mathsf{com}(g')} Q \xmapsto{\mathsf{norm}(K_0 \cdot g')} \xmapsto{\mathsf{norm}(K_1' \cdot K_2)} R \tag{4.7}$$

which gives the completion from Q, while Ψ gives the linearisation. □

From this follows that any cas-queue process can be normalised:

Corollary 4.12 (normalisability). *For a* cas*-queue process P, $P \xmapsto{\mathsf{nc}}{}^{*} P'$ such that P' is in normal form, where we write $P \xmapsto{\mathsf{nc}} Q$ when $P \xmapsto{\mathsf{nc}(g)} Q$ for some g.*

Normalisability implies linearisability because a normal form can always be reached from an initial process by a "linearised" sequence (where threads are not interleaved).

Now construct a relation $\mathcal{R}_{\mathsf{cas}}$ between cas-queue processes and the derivatives of $\mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset \rangle)$ as follows: (1) $\mathsf{CQemp}(r)\ \mathcal{R}_{\mathsf{cas}}\ \mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset \rangle)$; (2) If $P\ \mathcal{R}_{\mathsf{cas}}\ q$, and either $P \overset{\ell}{\longmapsto} P'$ and $q \xrightarrow{\ell} q'$, or $P \xmapsto{\mathsf{com}(g)} P'$ and $q \xrightarrow{\mathbf{com}(g)} q'$, then $P'\ \mathcal{R}_{\mathsf{cas}}\ q'$. (3) If $P\ \mathcal{R}_{\mathsf{cas}}\ q$ and $P \xmapsto{\mathsf{nc}} P'$ then $P'\ \mathcal{R}_{\mathsf{cas}}\ q$. In $\mathcal{R}_{\mathsf{cas}}$, all non-commit τ-actions are associated with the non-action in abstract queues.

Since *every* one-step transition in abstract queues is $\approx$-state-changing, $\mathcal{R}_{\mathsf{cas}}$ may be regarded as an abstract statement for atomicity and, therefore, linearisability. By Prop. 4.12, we can normalise each $\mathcal{R}_{\mathsf{cas}}$-related cas-queue process, noting that a cas-queue process in normal form has the same action capability as the related abstract queue. Then we define

an auxiliary relation $\widehat{\mathcal{R}_{\mathsf{cas}}}$, which maps normal forms to abstract queues by extracting the queue information from the former, as follows.

$$P \,\widehat{\mathcal{R}_{\mathsf{cas}}}\, p \quad \stackrel{\text{def}}{\Leftrightarrow} \quad P \text{ normal form and } p = \mathsf{AQ}(r, \langle \mathbf{req}(P), \mathbf{val}(P), \mathbf{ans}(P)\rangle)$$

where the functions $\mathbf{req}(-)$, $\mathbf{val}(-)$ and $\mathbf{ans}(-)$ are defined on cas-queue processes in normal forms as follows. Let $P = (\nu h, t, nd_0, \ldots, nd_n)(\mathsf{CQ}(r, h, t) \mid LL_P \mid \prod_{1\le i\le m} P_i)$ be a cas-queue process in normal form. Then:

- Let $1 \le i \le m$, such that P_i is a pending process. We first define $\mathbf{req}(P_i)$, as follows:
  - If P_i is of the form $\mathsf{EnqCom}_{\mathsf{cas}}\langle \ldots, v, g\rangle$, then $\mathbf{req}(P_i) = \mathtt{enq}(v, g)$;
  - On the other hand, if P_i is either $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle \ldots, g\rangle$ or $\mathsf{DeqCom}_{\mathsf{cas}}\langle \ldots, g\rangle$, then $\mathbf{req}(P_i) = \mathtt{deq}(g)$.

  Then $\mathbf{req}(P)$ is the set $\{\mathbf{req}(P_i) \mid 1 \le i \le m \ \ \wedge \ \ P_i \text{ is a pending process}\}$.
- Let LL_P be of the form:

  $$\mathsf{Ptr}(h, nd_H, ctr_H)|\mathsf{Ptr}(t, nd_T, ctr_T)|\mathit{\Pi}_{0\le i\le n}(\mathsf{Node}(nd_i, v_i, nxt_i) \mid \mathsf{Ptr}(nxt_i, nd_{i+1}, ctr_i))$$

  then $\mathbf{val}(P) = v_{H+1}, \ldots v_n$.
- Finally $\mathbf{ans}(P) = \{P_i \mid 1 \le i \le m \ \ \wedge \ \ P_i \text{ is an answer}\}$.

This extraction of information from the syntax makes reasoning very simple. Then it is good to use $\widehat{\mathcal{R}_{\mathsf{cas}}}$ in place of $\mathcal{R}_{\mathsf{cas}}$ when possible. The following lemma establishes a bridge between the two relations:

Lemma 4.13. *Let P be a* cas*-queue process in normal form. Then:*

1. *$P\ \mathcal{R}_{\mathsf{cas}}\, p$ implies $P\,\widehat{\mathcal{R}_{\mathsf{cas}}}\,p$.*
2. *If $P\ \widehat{\mathcal{R}_{\mathsf{cas}}}\, p$ then $P \stackrel{\ell}{\mapsto}$ iff $p \stackrel{\ell}{\rightarrow}$, with ℓ being a commit or visible.*

Proof of Lemma 4.13(1) It can be easily shown (Lemma G.17) that:

$$\Phi : \mathsf{CQemp}(r) \stackrel{\ell_1}{\mapsto} \ldots \stackrel{\ell_n}{\mapsto} P \qquad \Psi : \mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset\rangle) \stackrel{\widehat{\ell_1}}{\longrightarrow} \ldots \stackrel{\widehat{\ell_n}}{\longrightarrow} p$$

The proof consists of three steps: 1) we linearise Φ; 2) we simulate the linearised sequence from $\mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset\rangle)$, reaching an abstract queue p', s.t. $P\,\widehat{\mathcal{R}_{\mathsf{cas}}}\,p'$; 3) we show $p = p'$.

Since P is in normal form, it follows from Proposition 2.3(3) and from Lemma 4.11, that we have:

$$\Phi' : \mathsf{CQemp}(r) \xrightarrow{s} \mapsto^{\mathsf{norm}(K)} \mapsto^{\mathsf{norm}(K')} \xrightarrow{s'} P$$

where: s is a sequence of all the inputs and s' is a sequence of all the outputs that appear in Φ; K is the sequence of names of the threads which have committed in Φ (in the same order) and K' contains the names of the remaining threads. The whole sequence can be simulated from $\mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset\rangle)$, omitting non-commit τ-actions. Let p' be the abstract queue obtained after this simulation. Note that:

1. each input adds a selection on r to the queue process and the corresponding request to the abstract queue;

2. each local normalisation in the partition K reduces a selection on r in the queue process, performs the requested operation (by adding a new node to the linked list if it is an enqueue, by swinging the head pointer forward if it is a dequeue) and produces an answer on the requested continuation name;
3. since only the commit action is simulated, the simulation of a local normalisation in the partition K consists of a single τ-transition which removes the corresponding request, modifies the queue accordingly (dequeue or enqueue) and produces the same answer;
4. the local normalisations in the partition K' are not simulated by the abstract queue;
5. each output transition removes the same answer from both the queue process and the abstract queue.

Then $p' = \mathsf{AQ}(r, \langle \mathbf{req}(P),\ \mathbf{val}(P),\ \mathbf{ans}(P)\rangle)$ and $P\,\widehat{\mathcal{R}_{\mathsf{cas}}}\,p'$.

Now we show that $p = p'$. Let $\Psi' : \mathsf{AQ}(r, \langle \emptyset,\ \varepsilon,\ \emptyset\rangle) \xrightarrow{t} p'$ be the transition sequence simulating the linearised sequence from the abstract queue. Note that Ψ and Ψ' contain: 1) the same inputs; 2) the same outputs; 3) the same commits (in the same order). Now, for any request in p, a corresponding input must have occurred in Ψ, while a corresponding commit may not have occurred. Since the same holds for Ψ', that request must be also in p'. A similar reasoning applies to outputs and queue values. Note also that queue values must occur in the same order in p as in p', since the commits occurred in the same order in Ψ as in Ψ'. Then $p = p'$ and $P\,\widehat{\mathcal{R}_{\mathsf{cas}}}\,p$. □

Proof of Lemma 4.13(2) Note first of all that P is in normal form, since the relation $\widehat{\mathcal{R}_{\mathsf{cas}}}$ is only defined on queue processes in normal form. Then let $\{P_1, \ldots P_m\}$ be the set of all the pending processes and answers of P. We prove both directions by cases of ℓ.

– Let ℓ be an input transition label on r, of either form:

$$r\&enq(v, g) \qquad \text{or} \qquad r\&deq(g)$$

By definition an ℓ-labelled transition is always possible, both in queue processes and in abstract queues. Then $P \overset{\ell}{\longmapsto}$ if and only if $p \xrightarrow{\ell}$. Note also that, even though in the untyped LTS P admits input transitions on channels different from r, this possibility is prevented in the typed LTS.

– Now let ℓ be a commit action label $\mathbf{com}(g)$, for some continuation name g. Since P is in normal form,

$$P \overset{\mathsf{com}(g)}{\longmapsto} \iff g \text{ appears free in a pending process } P_i \text{ of } P$$

The pending process P_i is associated to a request by $\mathbf{req}(P)$, where: $\mathbf{req}(P_i) = \mathtt{enq}(v, g) \in \mathbf{req}(P)$, if P_i is of the form $\mathsf{EnqCom}_{\mathsf{cas}}\langle \ldots, v, g\rangle$; and $\mathbf{req}(P_i) = \mathtt{deq}(g) \in \mathbf{req}(P)$, if P_i is of either form $\mathsf{DeqCom}_{\mathsf{cas}}\langle \ldots, g\rangle$ or $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle \ldots, g\rangle$. So let $\mathbf{req}(P_i) = \mathtt{req}(g) \in \{\mathtt{enq}(v, g) \mid v \text{ is a value}\} \cup \{\mathtt{deq}(g)\}$. We have:

$$g \text{ appears free in a pending process } P_i \text{ of } P \iff \mathtt{req}(g) \in \mathbf{req}(P)$$

Note that $p = \mathsf{AQ}(r, \langle \mathbf{req}(P),\ \mathbf{val}(P),\ \mathbf{ans}(P)\rangle)$, since $P\,\widehat{\mathcal{R}_{\mathsf{cas}}}\,p$. Then, by the definition of the LTS for abstract queues, we have:

$$\mathtt{req}(g) \in \mathbf{req}(P) \iff p \xrightarrow{\mathbf{com}(g)}$$

Then $P \overset{\mathsf{com}(g)}{\longmapsto}$ if and only if $p \xrightarrow{\mathbf{com}(g)}$.

– Finally, consider the output case. Let ℓ be a an output transition label on a continuation name g: $\ell = \overline{g}\langle v\rangle$, where v is a value (including $\vec{v}$, the empty vector and $null$). The proof for this case follows a similar reasoning as the previous one:

$$\begin{array}{lll} P \overset{\overline{g}\langle v\rangle}{\longmapsto} & \iff & \overline{g}\langle v\rangle \text{ is an answer of } P \qquad \text{By Def. of norm. form} \\ & \iff & \overline{g}\langle v\rangle \in \mathbf{ans}(P) \qquad \text{By Def. of } \mathbf{ans}(P) \\ & \iff & p = \mathsf{AQ}(r, \langle \mathbf{req}(P),\, \mathbf{val}(P),\, \mathbf{ans}(P)\rangle) \overset{\overline{g}\langle v\rangle}{\longrightarrow} \qquad \text{By Def. of the LTS} \end{array}$$

Then $P \overset{\overline{g}\langle v\rangle}{\longmapsto}$ if and only if $p \overset{\overline{g}\langle v\rangle}{\longrightarrow}$. □

With these two key result we can establish the bisimilarity of $\mathcal{R}_{\mathsf{cas}}$:

Proposition 4.14. *1.* $\mathcal{R}_{\mathsf{cas}}$ *is a weak bisimulation.*
2. $\mathsf{CQemp}(r) \approx \mathsf{AQemp}(r)$.

Proof. (2) is an immediate corollary of (1), so we show (1). Assume $P\ \mathcal{R}_{\mathsf{cas}}\, p$. By Corollary 4.12, we know $P \overset{\mathsf{nc}}{\longmapsto}{}^{*}\ P_0$ such that P_0 is in normal form. By the construction of $\mathcal{R}_{\mathsf{cas}}$, we have $P_0\ \mathcal{R}_{\mathsf{cas}}\, p$. By Lemma 4.13 (1), we have $P_0\ \widehat{\mathcal{R}_{\mathsf{cas}}}\, p$. First assume $P \overset{\ell}{\longmapsto} P'$. If ℓ is a non-commit τ-action then $P'\ \mathcal{R}_{\mathsf{cas}}\, p$ by construction. If ℓ is a commit τ-action then $P_0 \overset{\ell}{\longmapsto}$ (since the thread is already pending in P it should be pending in P_0). By Lemma 4.13 (2), $p \overset{\ell}{\rightarrow} p'$ for some p'. By the construction of $\mathcal{R}_{\mathsf{cas}}$, we have $P'\ \mathcal{R}_{\mathsf{cas}}\, p'$, as required. Same reasoning if ℓ is an output. The case of an input is immediate. For the other direction, assume $p \overset{\ell}{\rightarrow} p'$. ℓ cannot be a non-commit τ. Let it be a commit or visible. Then by Lemma 4.13 (2), $P_0 \overset{\ell}{\longmapsto} P'$ for some P'. That is, $P \overset{\mathsf{nc}}{\longmapsto}{}^{*} \overset{\ell}{\longmapsto} P'$. By construction of $\mathcal{R}_{\mathsf{cas}}$, we have $P'\ \mathcal{R}_{\mathsf{cas}}\, p'$, as required. □

Lock-based Queues. The proofs of the functional correctness for the lock-based implementation is simpler, which is left to § G, obtaining:

Proposition 4.15.
$\mathsf{LQemp}(r) \approx \mathsf{AQemp}(r)$.

Combining Propositions 4.14 and 4.15, we obtain:

Theorem 4.16 (functional correctness of cas-queues and lock-queues).
$\mathsf{CQemp}(r) \approx \mathsf{LQemp}(r) \approx \mathsf{AQ}(r, \langle \emptyset,\, \varepsilon,\, \emptyset\rangle)$.

5 Application: Semantic Separation of Queues (2)

5.1 Global Progress

We now move to the global progress properties. We focus on the proof of non-blockingness of $\mathsf{CQemp}(r)$, the case of $\mathsf{LQemp}(r)$ is briefly discussed at the end and treated in detail in § G.8. In order to ensure non-blockingness, we need to ensure resilience first. Resilience is implied by the following key result in which linearisability (Lemma 4.11) plays again a central role.

Proposition 5.1. *Let* P *be a* cas*-queue process and let* $\Phi : \mathsf{CQemp}(r) \longrightarrow P$. *Then an output on a name* g *is blocked in* Φ *only if the* g*-thread of* Φ *contains a failing reduction.*

Proof. By Lemma G.29, we can assume that P is in general form,

$$P \;\equiv\; (\nu h, t, nd_0..nd_n)(\; \mathsf{CQ}(r, h, t) \;\mid\; \textstyle\prod_{1\leq i\leq m} P_i \;\mid\; LL \;)$$

Since $c \notin \texttt{allowed}(\Gamma_Q)$, c can only be allowed in Γ_P if an input occurred in Φ carrying c as continuation name and generating an enqueue/dequeue request, of either form $\mathsf{EnqReq}_{\text{cas}}\langle r, v, c\rangle$ or $\mathsf{DeqReq}_{\text{cas}}\langle r, c\rangle$. Furthermore, the name c may not disappear from the process until an output on c occurs, as the reductions of the above requests show. However, an output on c may not have occurred in Φ because $c \in \texttt{allowed}(\Gamma_P)$. Then, since the c-thread is non-failing in Φ, c must appear in a thread P^c of P which is in LMF.

Now we can use Lemma 4.11 again, which implies that there is $\Psi : \Gamma_P \vdash P \longrightarrow \Gamma'_P \vdash P'$, where P^c is reduced to an answer, i.e. an output with subject c. Then we can perform an output on c as the next step. Then $c \notin \texttt{blocked}(\Phi)$. □

Note that a finitely failing fair sequence Φ consists of a finitely failing prefix Φ_f and of a non-failing postfix Φ_{nf}. Then Proposition 5.1 has the following corollary.

Corollary 5.2. *Let Φ be a finitely failing fair transition sequence from a queue process and let $\Gamma \vdash P$ be in the non-failing postfix of Φ. If a g-thread is non-failing up to $\Gamma \vdash P$, either it terminates successfully or has infinite τ-transitions in Φ.*

Finally:

Proposition 5.3 (NB). $\mathsf{CQemp}(r)$ *is NB.*

Proof. Resilience is from Prop. 5.1. Let $\Gamma \vdash P$ occur in a finitely failing fair sequence Φ from $\mathsf{CQemp}(r)$ and let Φ_{nf} be the post-fix of Φ from $\Gamma \vdash P$. W.l.o.g. let Φ_{nf} be non-failing. By contradiction, assume that $\texttt{allowed}(\Gamma) \setminus \texttt{blocked}(\Phi) \neq \emptyset$ and that no non-failing thread of Φ commits in Φ_{nf}. Then by Corollary 5.2, any non-failing thread contains infinitely many τ-transitions. Since no other thread commits, neither the linked-list sub-process nor the head and tail pointers are ever affected. Then, by fairness, some non-failing thread reaches its pending LNF and commits in Φ_{nf}: a contradiction. Hence, if $\texttt{allowed}(\Gamma) \setminus \texttt{blocked}(\Phi) \neq \emptyset$ at least one thread commits and, by fairness, it reaches an answer and outputs. □

Note also that $\mathsf{CQemp}(r)$ is *not* WF, since we may not ensure that all the threads, in every execution, end up committing. Instead, $\mathsf{LQemp}(r)$ is not resilient, hence it is not NB either. This is because a failure in the critical section by one thread would block all other threads. But in the absence of failures, every thread in a transition sequence from $\mathsf{LQemp}(r)$ can enter the critical section by fairness (more details in § G.8).

To summarise:

Theorem 5.4 (global progress properties of cas-queues and lock-queues). $\mathsf{CQemp}(r)$ *is NB but not WF and* $\mathsf{LQemp}(r)$ *is WWF but not NB.*

5.2 Fair Preorder and Separation

We now define a fair pre-order which can semantically separate, among others, $\mathsf{CQemp}(r)$ and $\mathsf{LQemp}(r)$. Recall from § 3 that $\mathsf{FT}(\Gamma \vdash P)$ denotes the set of fair transition sequences with at most finite failures from $\Gamma \vdash P$. Below $\widehat{\Phi}$ denotes the result of abstracting away all τ-actions from the (possibly infinite) action label sequence of Φ. Then we define:

$$\mathsf{WFT}(\Gamma \vdash P) \stackrel{\text{def}}{=} \{\langle \widehat{\Phi}, \mathtt{blocked}(\Phi)\rangle \mid \Phi \in \mathsf{FT}(\Gamma \vdash P)\}$$

Above $\widehat{\Phi}$ is needed because we wish to know what visible sequences are possible in the presence of stalled threads; and $\mathtt{blocked}(\Phi)$ is needed because we wish to know exactly which threads are stalled besides those in which a failing reduction occurs. We define the fair pre-order (the notion "weak bisimulations" below is directly from Definition 2.4).

Definition 5.5 (fair preorder and fair bisimilarity). A *fair preorder* $\mathcal{R}$ is a weak bisimulation s.t. $\mathsf{WFT}(\Gamma \vdash P) \supseteq \mathsf{WFT}(\Gamma \vdash Q)$, for any $\Gamma \vdash P \mathcal{R} Q$. A *fair bisimulation* is a symmetric fair preorder, $\precsim_{\mathsf{fair}}$ is the maximum fair preorder and $\approx_{\mathsf{fair}}$ the maximum fair bisimulation. $\Gamma \vdash P \precnsim_{\mathsf{fair}} Q$ stands for $P \precsim_{\mathsf{fair}} Q$ but *not* $P \approx_{\mathsf{fair}} Q$.

For instance, $\mathsf{Lck}^{\mathsf{spin}}\langle u\rangle \approx \mathsf{Lck}\langle u\rangle$ (both defined in § 3) but $\mathsf{Lck}^{\mathsf{spin}}\langle u\rangle \precnsim_{\mathsf{fair}} \mathsf{Lck}\langle u\rangle$, since a fair sequence from the former allows useless infinite looping of a thread.

The fair pre-order ensures preservation of the global progress properties.

Proposition 5.6 (preservation of WF and NB). *If $P \precsim_{\mathsf{fair}} Q$ and $P \in$ **NB** (resp. $P \in$ **WF**), then $Q \in$ **NB** (resp. $Q \in$ **WF**).*

Proof. We only show the proof for wait-freedom, the same reasoning applies also to non-blockingness.

Let Γ be the typing of both P and Q. Let $\Phi' : \Gamma \vdash Q \longrightarrow \Gamma' \vdash Q'$ and let s be the sequence of labels appearing in Φ'. Let Φ'' be a non-failing fair transition sequence from $\Gamma' \vdash Q'$ such that a process $\Gamma'' \vdash Q''$ appears in Φ'' and $c \in \mathtt{allowed}(\Gamma'') \setminus \mathtt{blocked}(\Gamma' \vdash Q')$. Let t be the sequence of labels appearing in Φ'' up to $\Gamma'' \vdash Q''$ and let u be the sequence of labels appearing in Φ'' after $\Gamma'' \vdash Q''$. Note that the concatenation of Φ' and Φ'' is also fair.

Since $WFT(\Gamma \vdash Q) \subseteq WFT(\Gamma \vdash P)$, there is a fair transition sequence Ψ from $\Gamma \vdash P$ whose sequence of labels after abstracting non-failing τ-transitions becomes $\hat{s}\hat{t}\hat{u}$. Then let Ψ decompose as follows: $\Psi' : \Gamma \vdash P \longrightarrow \Delta' \vdash P'$, $\Psi'' : \Delta' \vdash P' \longrightarrow \Delta'' \vdash P''$ and $\Psi''' : \Delta'' \vdash P'' \longrightarrow$; where $\hat{s}$ appears in Ψ', $\hat{t}$ appears in Ψ'' and $\hat{u}$ appears in Ψ'''. Then $\mathtt{blocked}(\Delta' \vdash P') = \mathtt{blocked}(\Gamma' \vdash Q')$, because they both started from Γ and because the same input, output and failing transitions occurred since then (in the same order). Similarly $\mathtt{allowed}(\Delta'') = \mathtt{allowed}(\Gamma'')$. Hence, $c \in \mathtt{allowed}(\Delta'') \setminus \mathtt{blocked}(\Delta' \vdash P')$. But since $\Gamma \vdash P$ is wait free, an output on c occurs in Ψ'''. Then an output on c occurs in Φ'' after $\Gamma'' \vdash Q''$ as well. Then $\Gamma \vdash Q$ is wait free. □

Now, we show a strict inclusion of the weak finitely failing fair traces of cas-based and lock-based queue processes (§ 4). The following lemma facilitates the reasoning.

Lemma 5.7. $\mathsf{WFT}(P) \subseteq \mathsf{WFT}(Q)$ *iff it is so w.r.t. molecular action sequences.*

Molecular action sequences are used on both sides in the following strict inclusion proof:

Lemma 5.8. $\mathsf{WFT}(\mathsf{CQemp}(r)) \subseteq \mathsf{WFT}(\mathsf{LQemp}(r))$.

Intuitively, we only need to map a fair finitely failing trace of a cas-queue process to an analogous trace of a lock-queue process related by the bisimulation of Theorem 4.16, where a failure in the former is mapped to a failure *before* lock acquisition in the latter.

As for the separation result, we only need to show a weak fair finitely failing trace of $\mathsf{LQemp}(r)$ not belonging to $\mathsf{CQemp}(r)$. Suppose a thread fails in the critical section and then infinitely many other requests come, generating threads which may only progress up to some point and then are blocked. Such a sequence is finitely failing and it is also fair: in particular it is maximal, because no thread can progress further. It can be shown that a sequence with the same external behaviour and set of blocked outputs is not possible from $\mathsf{CQemp}(r)$, where every blocked output corresponds to a failing reduction (hence infinitely many blocked outputs imply infinitely many failing reductions). Similarly we can easily prove

The bisimilarity part of the proof comes from Theorem 4.16. We conclude:

Theorem 5.9. $\mathsf{LQemp}(r) \precnsim_{\mathsf{fair}} \mathsf{CQemp}(r)$.

6 Strict Resilience and Obstruction-Freedom

As further results attesting the uniformity of our framework, we compare strict resilience and *obstruction-freedom* [12], which ensures the completion of an operation if it is performed in isolation. For the following comparison, we assume the notions of thread and of threaded process to be as defined below. Recall that by *derivative* we mean a multi-step labelled transition derivative, cf. § 3. Also, an *active action* is an output or a conditional which does not occur under an input prefix or a conditional. A τ-redex is either a pair of an input and an output which can induce a τ-action (reduction) or an active conditional (which can thus be reduced as well, also inducing a τ-action). An output-redex is an active output action which can be outputted by the underlying typing (i.e. its subject is "allowed"). Finally, a redex is either a τ-redex or an output redex.

Below, note that the occurrence of an input transition (in general) adds a redex to the receiving process; while the reduction of a redex spawns a new sub-process, possibly inducing one or more new redexes.

Definition 6.1 (thread, thread redex, thread sub-process). Let $\Phi : P_0 \xrightarrow{\ell_0} P_1 \xrightarrow{\ell_1} P_2 \xrightarrow{\ell_2} \ldots$. A *thread* η_Φ of Φ is a sequence of redexes which appear in successive derivatives of Φ, such that:

1. the first redex in η_Φ is a redex of P_k, for $k \geq 1$, which has been induced by the input transition $P_{k-1} \xrightarrow{\ell_{k-1}} P_k$;
2. the other redexes appear in later derivatives such that, for $i > 1$, the i-th redex has been induced by the reduction of the j-th redex, for some $j < i$.

We may omit the indexing transition sequence name when not relevant and just write η. We may also say that η is a thread from P_i (for any $i < k$).

A *redex of* η is any redex that appears in η. The *sub-process of* η *at* P_j consists of all the sub-processes of P_j which have been spawned by reductions of redexes of η. We may also indicate a sub-process of η (omitting the derivative in which it appears).

The initial input starting a thread may be considered as the request of an operation: i.e. an enqueue or a dequeue operation in the case of cas-queue processes. And generally, the following reduction spawns a sub-process like those defined for enqueue and dequeue.

Below we use the term "pending" in the sense of (3.1) on Page 9, i.e. when an action appears whose subject is an allowed output channel on which no output has occurred yet. To start with, we consider processes which do not have allowed output channels, so that they may not contain any free output: a free output can only become available if it is introduced from the outside. Without loss of generality, we still follow the assumption of Convention 4.1, that all inputs of names are bound inputs (this ensures that every time an output is introduced, it is fresh). We can now define threaded processes:

Definition 6.2 (threaded process). Let $\Gamma \vdash P$, where no output is allowed by Γ. We say that P is *threaded* when any thread η from P is such that:

1. the reduction of any τ-redex of η induces exactly one other redex;
2. exactly one enabled output channel appears in each sub-process of η.

The first condition above enforces sequentiality inside each thread. It also enforces a certain shape to threads, by which a thread starts with a sequence of τ-redexes and possibly ends with an output redex, such that: the first redex has been induced by the initial input transition and the reduction of each τ-redex induces the following one in the sequence; an output ends a thread because it does not induce any more redexes. The second condition reflects the intuition that, in our observational framework, a thread is supposed to signal completion by emitting an output. Note however that this condition alone does not imply that the output will occur. Note also that the assumption that all inputs are bound ensures that each enabled output appears in the sub-process of just one thread (at any time).

The standard presentation of shared variable algorithms focuses on what amounts to threaded processes. For example, the process encoding of queues given in § 4 has this structure. We now prove a basic characterisation result.

Definition 6.3 (obstruction-freedom). We say a threaded P is *obstruction-free* if the following holds: for each maximal possibly failing Φ from P, such that P' occurs in Φ and, after P', a single thread in P' is reduced contiguously and without failures; then eventually an output occurs (from P').

Proposition 6.4 (characterisation of obstruction freedom). *A threaded P is obstruction free if and only if it is strictly resilient.*

Proof. Suppose P is obstruction-free. Take a finitely failing and fair Φ from P. By Definition 6.3, if some thread η does not fail, it can output in Φ. Then the unique output channel appearing in a(ny) sub-process of η is not blocked. Then the number of blocked outputs $|\mathtt{blocked}(\Phi)|$ may not be bigger than the number of failures in Φ. Then P is strictly resilient.

Next suppose P is strictly resilient. Take P' as in Definition 6.3 and make every thread of P' fail, except for a single thread P_c. Since we want to reduce P_c contiguously, the failures in the other threads do not affect its future behaviour while they allow us to use the strict resilience condition. Let c be the unique enabled output channel in P_c. By definition of strict resilience c is *not* blocked (otherwise the number of blocked outputs would be greater than the number of failures, Definition 3.5). Since we made all the other

threads fail, and since P_c generates a unique sequence of τ-redexes (by Definitions 6.1 and 6.2), a unique execution path is possible. Hence, this path must contain an output at c (if not, c would be blocked, contradicting resilience). Hence P is obstruction-free, as required. □

Thus obstruction-freedom has now been articulated in a common framework based on blocking effects. As we explained in § 4, the explicit inclusion of resilience in the global progress properties is necessary in our setting, as it provides for a minimal safety condition. This makes the intensional version of the properties match the intuition that they should be stronger than obstruction-freedom. Other than that, their definitions are as expected, i.e. Definition 3.8: in non-blockingness, when an output is allowed and not blocked, some output will occur; in wait-freedom, when an output is allowed and not blocked, that output will occur.

Our definition of non-blockingness corresponds to the most general interpretation: not only we allow an unbounded number of request to arrive, but also an unbounded number of operations to be concurrently executed. In real-world environments, such as web servers with popular demands, ours is a practically relevant approach, since the number of future requests may be unpredictable and we wish to know how they will affect preceding actions. For instance, an extremely unfair execution consists of continuously receiving new requests which prevent any operation to be carried out.

Traditionally, algorithmic studies have focused on an arbitrary but bounded number of concurrently executing operations, where the above execution may not take place. For example, this is the setting in which the informal notion of non-blockingness quoted in § 1 is based on. However, such a view also limits the applicability of the definition: it has been shown that binding the number of concurrently executing operations is equivalent to binding the number of requests [10]. We see this as a limitation because, when only a finite number of requests arrive, non-blockingness and wait-freedom coincide: suppose n requests arrive, non-blockingness ensures that *one* of them will be answered; afterwards only $n - 1$ unanswered requests remain, so by applying the definition n times, all the requests will be eventually answered.

Rather than binding the number of concurrent operations, we require the scheduler to be fair, so that even if an infinite number of requests arrive, this does not prevent other operations from making progress. Now we formally show that our intensional non-blockingness strictly implies the finite requests definition as formalised below (the opposite is obviously not true because a definition that works for a bounded number of requests does not say anything in the case the number of requests is unbounded).

Definition 6.5 (thread-wise NB). Let P be threaded and suppose Φ is a finitely failing fair transition sequence of P such that only finitely many non-linear inputs take place. We say that P is *thread-wise non-blocking* if the following holds: if an output channel c is pending in a derivative P' of Φ and c appears in a sub-process of a non-failing thread η_Φ, then an output at c eventually occurs in Φ.

In a thread-wise non-blocking process, *each* (non-failing) thread can "complete its operation in a finite number of its own steps", as far as only a finite number of requests occur. This matches traditional informal definitions of non-blockingness, as explained above.

Proposition 6.6. *Suppose a threaded P is intensionally non-blocking. Then it is also thread-wise non-blocking.*

Proof. Let P be threaded and intensionally non-blocking. Let Φ be a finitely failing and fair transition sequence from P containing only finitely many inputs, let P' be a derivative of P occurring in Φ and let c be an output channel pending in P'. Since Φ has only finitely many inputs and only finitely many failures, we can assume w.l.o.g. that all the inputs and all the failures in Φ occur before P'. Then the set $A = \{a_1, a_2, .., a_n\}$ of pending output channels in P' is finite (note also $c \in A$). By intensional non-blockingness, an output shall take place at some channel of A, say a_i, in Φ after P'. Note that after an output occurs at an enabled channel, that channel is not enabled anymore. By applying the same procedure finitely many times, we are ensured that, within Φ, an output shall occur on each channel of A. In particular, we know that an output at c shall occur in Φ. By threadedness, we know this output comes from the thread introduced when c was received in an input. □

7 Related Work and Further Topics

7.1 Comparison With Previous Formalisations

The first to propose a formal characterisation of non-blockingness and wait-freedom was Dongol [6], who used an ad-hoc logical system, which may still be relatively unknown to a wider community. Later, Gotsman et al. [10] have given a formalisation in the better known separation logic combined with rely-guarantee assertions. Other works inspired by the above followed ([23] and [28]).

The current paper has taken a completely different approach. Rather than giving another logical formalisation, we have formalised the above properties in a process algebra (which is by itself a novelty). This naturally lead to the first extensional characterisation of these properties, captured only by observing the interactions between the system and its environment. There are no obligations on the environment, except for its obedience to the communication protocol (enforced by the typing system). In particular, the environment can request a new operation at any time. Hence, the number of concurrently executing operations is unbounded. This was not the case in the previous logical characterisations, where each process from a finite set was only able to start one operation at a time.

Nowadays, servers have to deal with an ever increasing number of requests and a model which binds the number of concurrently executing operations is not satisfactory. Consider a model that binds the number of concurrent operations and set this bound to K. Then consider a server which counts the number of operations it is performing, accepting and serving up to K requests concurrently and discarding the successive ones. According to the binding model, this server is wait-free, whereas the model presented in this paper correctly identifies it as non-blocking (to be precise, extensionally non-blocking, since discarded threads would be blocked without having failed).

Moreover, in our model we can represent any execution of any data structure. For instance, $\mathsf{CQemp}(r)$ may receive an unbounded number of enqueue requests as follows: after the first two requests, do a reduction in the second thread, then one in the first thread, then receive another request and repeat (since the first thread always follows the second, it will not be able to commit); once the second thread answers and the first one goes back to the beginning of the loop, repeat everything (where what is now the second thread was the third one in the previous round). This is an admissible execution in which the first thread never commits (while the others do). Since the number of threads is ever increasing, it cannot be represented in previous models.

With these two examples we have shown that our model is complete in the sense that it captures the behaviour of any data structure, representing any possible execution

(even under unfair schedulers, see below). Note also that, as [10] shows, if the number of concurrently executing operations is bounded, the non-blockingness proof can be reduced to the proof of termination of an execution with finitely many requests. At the same time, we have shown that if only finitely many requests come, non-blockingness implies wait-freedom. This does not mean that with bounded concurrency, non-blockingness implies wait-freedom, but it still gives an indication of the fact that unbounded concurrency is harder to prove.

The above distinctions apply to all of the cited papers, nonetheless we can say that, among those papers, [6] is the closest to our framework, since it takes failures into account, whereas [10] models failures by way of an unfair scheduler. Having an unfair scheduler forces atomic operations to be executed in a single step (which is not what happens in practice), whereas the combination of fairness and failures allows to obtain both the benefits of a fair scheduler (by specifying that a failure may not occur inside an atomic operation) and the ability to represent an unfair one (through failures).

While [6] manually forbids failures inside atomic operations, we forbid them by the use of linear types. Note also that Dongol only requires weak fairness: the reason why we require strong fairness is that the π-calculus model provides a finer representation, hence weak fairness in [6] is interpreted as strong fairness in our model. This is not constraining because, as we said, failures enable us to represent even unfair schedulers. Failures are exploited much further in our model: rather than just enabling them, we are interested in observing their effects on other threads, i.e. a failure in one thread may block another thread. This lead us to characterising resilience, which is one fundamental part of non-blockingness: any model with failures has to include some form of resilience. Implicitly, even the notion of non-blockingness in [6] does, by requiring that each failure only blocks the thread in which it occurs (strict resilience). We have further shown a whole class of resilience properties: by relaxing strict resilience, we obtain a whole range of new non-blocking properties. In the most relaxed notion of resilience, the exact number of failures is irrelevant. As a result, we obtain a fully observational characterisation of progress, allowing verification through compositional semantics.

7.2 Other Related Works

A key feature in our model is *fairness*, adapted from [4] to our asynchronous and typed framework (process typing inducing "enabled transitions"). The use of fairness to capture progress properties was first suggested in [31]. Later, other works have employed fairness in different calculi and to capture different progress properties than ours (cf. [3] and [5]).

Some of the previous formalisations of progress (liveness) properties in the π-calculus have also used fairness, including [1] and [17], whose common goal is to enforce liveness by typing. The merit of these and other type-based approaches to liveness is that it allows to statically ensure liveness. At the same time, it is hard to capture such global progress properties as non-blockingness and wait-freedom through static typing. Consider non-blockingness, which ensures that the output of *some* operation will eventually occur. Our theory correctly identifies all instances of non-blocking behaviours, while the type-based approach of [1, 17, 33] falls short of identifying such instances of non-blocking behaviours as Michael-Scott queue, since it enforces some kind of local progress at *every* operation. But it does not capture wait-freedom either, since such local progress is ensured by local causality alone, which may not imply completion of the overall operation. In particular,

the property called lock-freedom in [17] is not non-blockingness, as its term may suggest [12], but rather a locally ensured liveness property as in e.g. [33].

The correctness proofs of § 4 were inspired by notions from *linearisability*, such as *left-mover* and *right-mover* [14, 18]. Linearisability has already been applied to Michael-Scott's queue [2, 7, 32]. [11] shows non-blockingness is maintained when composed with linearisable libraries and [26] exploits modularity in the search for atomicity violations and reduces the state-explosion by requesting non-commuting operations before and after the one being tested. [30] also exploits modularity and refinement through the use of separation logic for verifying safety properties like atomicity.

We share some of the motivations with all of the works mentioned above, i.e. semantic understanding of non-blockingness, modularity, as well as exposing critical permutations. However, the formal framework is quite different. Apart from the introduction of a formal account of intensional and extensional global progress, the use of the π-calculus allows a uniform behavioural analysis at a very fine granularity level, including the use of local permutations through LTS for analysing linearisability. [11] does abstractly define linearisation, relying on separate tools for its concrete realisation. Their definition is based on a *begin-end* rather than on a *commit* (more fine-grained). [2] uses commits similar to, but coarser than, ours: instead of local permutations, they suspend execution of the simulating process B until the simulated one A commits. Then A is suspended and B starts and completes the operation. We believe our bisimilarity arguments semantically justify their automated proof by offering a behavioural characterisation of their transformations.

While the present work is focused on the extensional properties, the intensional formalisations can be exploited further, which may give a basis for a comprehensive semantic framework to analyse, specify, validate and classify both intensional and extensional properties of concurrent data structures, including proof-assistants (cf. [31, 32]), etc...

References

1. L. Acciai and M. Boreale. Responsiveness in process calculi. *TCS*, 409(1): 59–93. Elsevier. 2008.
2. D. Amit, N. Rinetzky, T. Rep, M. Sagiv and E. Yahav. Comparison under abstraction for verifying linearizability. *CAV'07*, 477–490. Springer, 2007.
3. S. Brookes. Deconstructing CCS and CSP: asynchronous communication, fairness, and full abstraction. *MFPS 16*. 2000.
4. D. Cacciagrano, F. Corradini and C. Palamidessi. Explicit fairness in testing semantics. *LMCS*, vol. 5(2:15), 27 pages. 2007.
5. 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. Springer. 2009.
6. B. Dongol. *Formalising progress properties of non-blocking programs.* In *ICFEM'06*, vol. 4260 of *LNCS*, 284–303. Springer, 2006.
7. I. Filipovic, P. W. O'Hearn, N. Rinetzky and H. Yang. Abstraction for concurrent objects. *ESOP'09*, 252–266. Springer. 2009.
8. N. Francez. *Fairness*. Springer, 1986.
9. Brian Goetz. *Java Concurrency in Practice.* Addison-Wesley, 2008.
10. A. Gotsman, B. Cook, M. Parkinson and V. Vafeiadis. Proving that non-blocking algorithms don't block. *POPL'09*, 16–28. ACM. 2009.
11. A. Gotsman and H. Yang. *Liveness-preserving atomicity abstraction. ICALP'11*, 453–465. Springer. 2011.
12. M. Herlihy, V. Luchangco and M. Moir. Obstruction-free synchronization: double-ended queues as an example. *ICDCS'03*, 522–529. IEEE Computer Society. 2003.

13. M. Herlihy and B. Sharit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2009.
14. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492. ACM. 1990.
15. K. Honda and N. Yoshida. A uniform type structure for secure information flow. *POPL'02*. 81–92. ACM. 2002.
16. N. Kobayashi, B.C. Pierce and D.N. Turner. Linearity and the Pi-calculus. TOPLAS, 21(5):914–947. ACM. 1999.
17. N. Kobayashi and D. Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. *TOPLAS*, 32(5:16). 49 pages. ACM. 2010.
18. E. Koskinen, M. Parkinson and M. Herlihy. Coarse-grained transactions. *POPL'10*. 19–30. ACM, 2010.
19. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564. ACM. 1978.
20. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Operating Systems Review*, 13(2):3–19. ACM. 1979.
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'96*. 267–275. ACM. 1996.
23. E. Petrank, M. Musuvathi and B. Steensgaard. Progress guarantee for parallel programs via bounded lock-freedom. *PLDI'09*. 144–154. ACM. 2009.
24. D. Sangiorgi. *πI: A Symmetric Calculus based on Internal Mobility.* In TAPSOFT'95, vol. 915 of *LNCS*, 172–186. Springer, 1995.
25. D. Sangiorgi. The name discipline of uniform receptiveness. TCS 221(1–2):457–493. Elsevier. 1999.
26. O. Schacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev and E. Yahav. Testing atomicity of composed concurrent operations. *OOPSLA'11*. 51–64. ACM. 2011.
27. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson–Prentice Hall, 2006.
28. B. Tofan, S. Bäumler, G. Schellhorn, W. Reif. *Temporal logic verification of lock-freedom.* In *MPC'10*, vol. 6120 of *LNCS*, 377–396. Springer, 2010.
29. M. Tokoro and V. Vasconcelos. *A Typing System for a Calculus of Objects.* In *ISOTAS'93*, vol. 742 of *LNCS*, 460–474. Springer, 1993.
30. A. Turon and M. Wand. A separation logic for refining concurrent objects. *POPL'11*. 247–258. ACM. 2011.
31. D. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1(3):273–292. Springer. 1989.
32. E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In *SoftMC'03*, vol. 89(3) of *ENTCS*, 450–463. Elsevier. 2010.
33. N. Yoshida, M. Berger and K. Honda. Strong Normalisation in the π-Calculus. *Information and Computation*, 191(2):145–202. Elsevier. 2004.

$$[\mathsf{Bra\text{-}L}]\frac{\Gamma^\star, \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&^L_{i\in I} l_i(\vec{\tau}_i) \vdash u\&^L_{i\in I} l_i(\vec{x}_i).P_i}$$

$$[\mathsf{Bra\text{-}\star}]\frac{\Gamma^{\uparrow\star}, u : \&^\star_{i\in I} l_i(\vec{\tau}_i), \vec{x}_j : \vec{\tau}_j \vdash P_j \ (\forall j \in I)}{\Gamma, u : \&^\star_{i\in I} l_i(\vec{\tau}_i) \vdash u\&^\star_{i\in I} l_i(\vec{x}_i).P_i}$$

$$[\mathsf{Sel}]\frac{\Gamma \vdash \vec{e} : \overline{\vec{\tau}_j} \ (j \in I) \quad Dom(\Gamma) = names(\vec{e})}{\Delta^{\uparrow\star}, \Gamma, u : \oplus_{i\in I} l_i(\vec{\tau}_i) \vdash \overline{u} \oplus l_j\langle\vec{e}\rangle}$$

$$[\mathsf{If\text{-}\star}]\frac{\Gamma^\star \vdash e : \mathsf{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q} \qquad [\mathsf{If\text{-}L}]\frac{\Gamma \vdash e^L : \mathsf{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathtt{if}^L\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q}$$

$$[\mathsf{Par}]\frac{\Gamma_i \vdash P_i (i = 1, 2)\ \Gamma_1 \asymp \Gamma_2}{\Gamma_1 \odot \Gamma_2 \vdash P_1 | P_2} \qquad [\mathsf{Res}]\frac{\Gamma, a : \tau^{\perp,\downarrow\star} \vdash P}{\Gamma \vdash (\nu a)P} \qquad [\mathsf{Inact}]\frac{}{\Gamma^\star \vdash \mathbf{0}}$$

$$[\mathsf{Rec}]\frac{\Gamma^\star, \vec{x} : \vec{\tau}, X : \vec{\tau} \vdash P \quad \Gamma \vdash \vec{e} : \vec{\tau}}{\Gamma \vdash (\mu X(\vec{x}).P)\langle\vec{e}\rangle} \qquad [\mathsf{Var}]\frac{\Gamma^\star \vdash \vec{e} : \vec{\tau}}{\Gamma, X : \vec{\tau} \vdash X\langle\vec{e}\rangle}$$

Fig. 3. Typing Rules for Processes

A How to Read this Appendix

The Appendices list auxiliary definitions (§ C and § D). Because of recent notational changes we are omitting the proofs in this version, to avoid *notational* inconsistencies.

B Auxiliary Definitions for Section 2

B.1 Linear Typing Rules

We summarise the linear type discipline used in the present inquiry. The construction follows [15, 33] except we do *not* suppress an L-output by an 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 ensure the properties of linear reductions (Proposition 2.3), as well as justifying the environment transitions (§ 2).

In the typing rules and in the following sections we shall follow the notation that a lack of modality in a type means both modalities apply, while a lack of annotation in a term may mean either that it is irrelevant (when both modalities apply) or that it is implicit from the context. Note in particular that the annotations in the terms are inferred from the modalities in the types. The typing rules are given in Figure 3, where "Γ, Δ" indicates the union of Γ and Δ as well as demanding their domains to be disjoint. In $\Gamma^\star$ (resp. in $\Gamma^{\uparrow\star}$), Γ can only contain non-linear (output) types and base types, including $\perp$. Finally, e^L means that all names appearing in e are L-annotated. We assume that each type is *input/output alternating*: in an input type $\&_{i\in I} l_i(\vec{\tau}_i)$, each τ_i is an output type, and dually.

In [Bra-L], an L-input can suppress free channels of all "possibly unavailable" modes, i.e. $\uparrow\star$ and $\downarrow\star$. A linear input has receptiveness [25], i.e. availability of input channels, by not being suppressed by another input. On the other hand, a non-linear input may be non-receptive. Note also that an $\uparrow L$ type is never suppressed under any input prefix.

In [Sel], the output rule is standard, dualising the carried types $\vec{\tau}_j$ in the premise. Note that Γ types all and only the names appearing in $\vec{e}$ (denoted in the rules by $names(\vec{e})$). Then we use Δ as a form of weakening for $\bot$ and non-linear output types.

The two rules for a conditional are standard except that, in [If-L], the guard must be L-annotated; while in [If] a conditional can only "suppress" channels of possibly unavailable modes (since the guard may be $\star$-annotated).

The rule [Par] for $|$ uses the relation $\asymp$ and operation $\odot$ from [15]. They are first defined on types, then extended to typings. Intuitively, on a linear channel, composition is allowed for at most one input and at most one output; on a non-linear channel, it is allowed for at most one input and zero or arbitrarily many outputs. Below note $\overline{\bot}$ is undefined.

1. $\tau \asymp \overline{\tau}$ always and $\tau \odot \overline{\tau} = \bot$ when the modality is linear, otherwise $\tau \odot \overline{\tau} = \tau$ if τ is an input type, and $\tau \odot \overline{\tau} = \overline{\tau}$ if τ is an output type.
2. $\tau \asymp \tau$ if τ is a non-linear output or 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, $\Gamma_1 \odot \Gamma_2$ is defined for each $u \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$, so that $\Gamma(u) = \Gamma_1(u) \odot \Gamma_2(u)$ if $u \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$, and otherwise types come from $\Gamma_{1,2}$.

In [Res], τ may only be $\bot$ or a non-linear input type. This is because $\boldsymbol{\nu}$-hiding is only possible for "self-contained" channels: for a linear input/output pair where composition has already occurred (type $\bot$) and for a non-linear input type, which can be associated to the type of a server. Finally, in [Inact], we allow $\mathbf{0}$ to be typed by possibly unavailable types.

The rules for recursion and variable, [Rec] and [Var], are standard.

C Auxiliary Definitions for Section 2

C.1 Linear Typing Rules

We summarise the linear type discipline used in the present inquiry. The construction follows [15, 33] except we do *not* suppress an L-output by an 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 ensure the properties of linear reductions (Proposition 2.3), as well as justifying the environment transitions (§ 2).

In the typing rules and in the following sections we shall follow the notation that a lack of modality in a type means both modalities apply, while a lack of annotation in a term may mean either that it is irrelevant (when both modalities apply) or that it is implicit from the context. Note in particular that the annotations in the terms are inferred from the modalities in the types. The typing rules are given in Figure 3, where "Γ, Δ" indicates the union of Γ and Δ as well as demanding their domains to be disjoint. In $\Gamma^\star$ (resp. in $\Gamma^{\uparrow\star}$), Γ can only contain non-linear (output) types and base types, including $\bot$. Finally, e^L means that all names appearing in e are L-annotated. We assume that each type is *input/output alternating*: in an input type $\&_{i\in I} l_i(\vec{\tau}_i)$, each τ_i is an output type, and dually.

In [Bra-L], an L-input can suppress free channels of all "possibly unavailable" modes, i.e. $\uparrow\star$ and $\downarrow\star$. A linear input has receptiveness [25], i.e. availability of input channels, by not being suppressed by another input. On the other hand, a non-linear input may be non-receptive. Note also that an $\uparrow L$ type is never suppressed under any input prefix.

In [Sel], the output rule is standard, dualising the carried types $\vec{\tau}_j$ in the premise. Note that Γ types all and only the names appearing in $\vec{e}$ (denoted in the rules by $names(\vec{e})$). Then we use Δ as a form of weakening for $\bot$ and non-linear output types.

The two rules for a conditional are standard except that, in [If-L], the guard must be L-annotated; while in [If] a conditional can only "suppress" channels of possibly unavailable modes (since the guard may be $\star$-annotated).

The rule [Par] for $|$ uses the relation $\asymp$ and operation $\odot$ from [15]. They are first defined on types, then extended to typings. Intuitively, on a linear channel, composition is allowed for at most one input and at most one output; on a non-linear channel, it is allowed for at most one input and zero or arbitrarily many outputs. Below note $\overline{\bot}$ is undefined.

1. $\tau \asymp \overline{\tau}$ always and $\tau \odot \overline{\tau} = \bot$ when the modality is linear, otherwise $\tau \odot \overline{\tau} = \tau$ if τ is an input type, and $\tau \odot \overline{\tau} = \overline{\tau}$ if τ is an output type.
2. $\tau \asymp \tau$ if τ is a non-linear output or 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, $\Gamma_1 \odot \Gamma_2$ is defined for each $u \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$, so that $\Gamma(u) = \Gamma_1(u) \odot \Gamma_2(u)$ if $u \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$, and otherwise types come from $\Gamma_{1,2}$.

In [Res], τ may only be $\bot$ or a non-linear input type. This is because $\boldsymbol{\nu}$-hiding is only possible for "self-contained" channels: for a linear input/output pair where composition has already occurred (type $\bot$) and for a non-linear input type, which can be associated to the type of a server. Finally, in [Inact], we allow $\mathbf{0}$ to be typed by possibly unavailable types.

The rules for recursion and variable, [Rec] and [Var], are standard.

D Auxiliary Definitions for Section 4

D.1 Typing Abstract Queues

Below we define the typing for abstract queues (which was briefly illustrated with an example in § 4.1), and discuss key properties of the induced typed transition. For simplicity we only treat abstract queues which store base type values such as natural numbers or booleans, though the typing and associated results can be easily extended to the case of a queue storing channels or composite values. Similarly for lock-based and cas-based queues.

Definition D.1 (typing for abstract queues). Let p consist of:

1. subject r;
2. set of requests $\{\mathtt{enq}(v_i, g_i)\}_{i \in I} \cup \{\mathtt{deq}(g_j)\}_{j \in J}$;
3. value sequence $v_1 \cdots v_n$;
4. set of answers $\{\overline{g_k}\langle\varepsilon\rangle\}_{k \in K} \cup \{\overline{g_l}\langle v_l\rangle\}_{l \in L}$.

where:

- I, J, K and L are mutually disjoint and $g_a \neq g_b$ whenever $a \neq b$, for $a, b \in I \cup J \cup K \cup L$;
- each v_i and each value from $\vec{v}_k$ is typable with the same base type, α (which we assume to include a special value ε, returned when the empty queue is dequeued).

Then we type p as follows:

$$r : \&^{\star}\{\texttt{enq}(\alpha \uparrow^{\star} ()), \texttt{deq}(\uparrow^{\star} (\alpha))\},\ \{g_i :\uparrow^{\star} ()\}_{i\in I\cup K},\ \{g_j :\uparrow^{\star} (\alpha)\}_{j\in J\cup K}\ \vdash\ p$$

We call $\Gamma \vdash p$ given in this way, *a typed abstract queue*.

Proposition D.2 (typed LTS over abstract queues). *Suppose $\Gamma \vdash p$ by the typing above and $p \xrightarrow{\ell} p'$ such that the bound names in ℓ are disjoint from* $\mathrm{dom}(\Gamma)$. *Then*

1. *If ℓ is an output or τ, then $\Gamma \xrightarrow{\ell} \Gamma'$ such that $\Gamma' \vdash p'$.*
2. *If ℓ is an input and $\Gamma \xrightarrow{\ell} \Gamma'$ for some Γ', then $\Gamma' \vdash p'$.*

We write $\Gamma \vdash p \xrightarrow{\ell} \Gamma' \vdash p'$ in these cases.

Proof. By easy inspection of each possible transition. Input and output are obvious, while we can immediately check the τ-action does not change the typing environment. □

Proposition D.2 defines the typed LTS over typed abstract queues. As a result, the same $\approx$ given for concrete processes can be defined over typed abstract queues, quotienting them in terms of their typed LTS.

Proposition D.3. *Suppose $\Gamma \vdash p \xrightarrow{\ell} \Delta \vdash q$. Then:*

1. *$\Gamma \vdash p$ and $\Delta \vdash q$ are not related by $\approx$.*
2. *Further p and q are not bisimilar under their untyped LTS.*

Proof. We just need to show (2) since it already implies (1). If ℓ is an input, it introduces a new request. Then q can do an action which is not possible from p. If ℓ is an output, it consumes an answer. Then q is no more able to do the same output. Now let $\ell = \tau$. In this case, before ℓ occurs we can receive another input, introducing a request for a new dequeue/enqueue operation. Then if this request is reduced first, the queue would be affected and ℓ may not be possible anymore, or it may end up producing a different answer. □

D.2 Lock-based Queue: Full Definition

General Structure. We recall from § 4 that the empty lock based queue has the following form:

$$\mathsf{LQemp}(r) \ \stackrel{\text{def}}{=}\ (\nu h, t, s, l)(\mathsf{Mtx}\langle l\rangle | \mathsf{LQ}(r, h, t, l) | \mathsf{LPtr}(h, s) | \mathsf{LPtr}(t, s) | \mathsf{LENode}(s, 0))$$

where

$$\begin{aligned}\mathsf{LQ}(r, h, t, l) \ \stackrel{\text{def}}{=}\ !\, r\&\{&enq(v, u) : (\bar{l}(g)g(y).P^{lck}_{enq}(v, t, y, u),\\ &deq(u) : (\bar{l}(g)g(y).P^{lck}_{deq}(h, t, y, u)))\}\end{aligned}$$

As we add new nodes, each node has the form $\mathsf{LNode}(r, v, nxt)$ which is a *non*-cas reference of the form $\mathsf{Ref}\langle r, \langle v, nxt\rangle\rangle$ (in practice we store a pair as a single value through encoding), where v is the value and nxt the name of the next node. When a node is inserted in the list, its successor value should always be $null$. For brevity, we write such a node with the empty successor, $\mathsf{LENode}(r, v)$, standing for $\mathsf{Ref}\langle r, \langle v, null\rangle\rangle$, as used in the empty queue above. It also uses head and tail pointers $\mathsf{LPtr}(r, nd)$ standing for a non-cas reference of the form $\mathsf{Ref}\langle r, nd\rangle$, where nd is the name of a node.

The enqueue and dequeue operations are given in the following subsection.

Enqueue and Dequeue Operations. The enqueue operation is simple (recall that the lock has already been acquired):

$$
\begin{array}{ll}
1 & P^{lck}_{enq}(v,t,y,u) = (\nu next)(\mathsf{LENode}(next,v)| \\
2 & \qquad (t \triangleleft read(tlNd). \\
3 & \qquad tlNd \triangleleft wrNxt\langle next\rangle. \\
4 & \qquad t \triangleleft write\langle next\rangle. \\
5 & \qquad (\bar{u}|\bar{y})))
\end{array}
$$

It creates a new node with reference $next$ and value v (line 1) then reads the name of the node contained in the tail t (line 2). Then it writes the name of the new node in the reference field of the last node (line 3) and swings the tail forward (line 4). Finally, it outputs through u and releases the critical section through y (line 5). The operation of the form $x \triangleleft read(y).P$ is syntactic sugar and has already been defined in § 4. $x \triangleleft write\langle y\rangle.P$ (on a pointer) is similarly defined and $x \triangleleft wrNxt\langle y\rangle.P$ (on a node) simply writes the node's pointer field, not the value field.

The dequeue is as follows:

$$
\begin{array}{ll}
1 & P^{lck}_{deq}(h,y,u) = h \triangleleft read(hdNd). \\
2 & \qquad t \triangleleft read(tlNd). \\
3 & \qquad \mathtt{if}^{L}\,(hdNd = tlNd)\ \mathtt{then} \\
4 & \qquad\quad \bar{y}|\bar{u}\langle null\rangle \\
5 & \qquad \mathtt{else}\ \ hdNd \triangleleft read(x,next). \\
6 & \qquad\quad h \triangleleft write\langle next\rangle. \\
7 & \qquad\quad \bar{y}|\bar{u}\langle x\rangle
\end{array}
$$

It first reads the head and the tail pointers (lines 1 and 2), then compares them (line 3). If they contain the name of the same node, it releases the lock through y and outputs $null$ through u (line 4). This ensures that the head never goes beyond the tail in the linked list. Otherwise, it reads the value of the node referred by the head as well as the name of its successor (line 5). It writes the name of the successor in the head pointer (line 6) and outputs the read value (line 7).

Commit Transitions. In § 4.3, we proposed a classification of internal actions in cas queue processes. In particular, we have identified those semantically state-changing τ-actions which we called *commits* and which should correspond to the internal actions that take place in the executions of abstract queues.

We can identify an analogous action in the execution of a lock-based queue as well. Below we identify the *commit transitions* in lock-based queues, denoted $\mathsf{com}(g)$ for a thread g. Note that each operation (enqueue or dequeue) should at some point perform one (and no more than one) commit transition.

We identify the commit τ-transition as the τ-action which acquires the lock: this action does not appear in $P^{lck}_{enq}(v,t,y,u)$, nor in $P^{lck}_{deq}(h,y,u)$, but appears as the first action in each branch of $\mathsf{LQ}(r,h,t)$. Since it is not followed by any linear reduction, this commit transition is also a *commit molecular action*, (i.e. a molecular action, § 4, which starts with a commit transition), denoted $\stackrel{\mathsf{com}(g)}{\longmapsto}$ and also called *commit*.

E Proofs for Section 2

In this appendix, we prove the main results of Section 2. They include:

– the subject transition (Proposition 2.3, page 6).
– properties of linear transitions (Proposition 2.3, page 6) and
– congruence of $\approx$ (Proposition 2.5, page 6)

After a preliminary in §E.1, we prove the above results in three subsections one by one.

E.1 Basic properties of typed processes

Below in (2) we say a channel is *active* if it occurs as subject *not* under an input or conditional. (3) means a sequence of linear reductions do not interfere with other reductions: they are semantically atomic.

Proposition E.1. *Let $\Gamma \vdash P$, where P is closed. Then:*

1. (subject reduction) *If $P \longrightarrow P'$ then $\Gamma \vdash P'$.*
2. (activeness) *If the mode of $\Gamma(c)$ is L and c is not an object in P, then $P \longrightarrow^* Q$ by linear reductions s.t. c is active in Q.*
3. (partial confluence [16]) *If $P \longrightarrow P_1$ is linear and $P \longrightarrow P_2 \not\equiv P_1$, then $P_1 \longrightarrow Q$ and $P_2 \longrightarrow Q$, the latter reducing the same redex as $P \longrightarrow P_1$.*

Following the standard routine, we first prove the Substitution and Weakening lemmas.

Lemma E.2 (substitution). *Let $\Gamma, x : \tau \vdash P$ Then:*

1. *if τ has a linear mode, for any $a \notin Dom(\Gamma)$ we have $\Gamma, a : \tau \vdash P\{a/x\}$.*
2. *otherwise, if $\Gamma \vdash e : \tau$ and $e \downarrow v$ we have $\Gamma \vdash P\{v/x\}$.*

Proof. The claim is trivially true for $P = \mathbf{0}$ and $X\langle\vec{e'}\rangle$. It is also true for $P = (\nu b)P_1$. Note that in this case x may not be b because we are assuming that x is free. The same applies to recursion and to both inputs. For inputs, if x is the subject then e must be a name a (no constant) since it has the same type as x. Moreover, if x is non-linear the subject maintains the same type it had in the premise: then the substitution applies because $\Gamma \vdash a : \tau$, by assumption. Instead, if x is linear, its typing does not appear in Γ and neither does a's (by assumption): then the substitution applies. Same reasoning for the output, both as subject and as object.

Assume that the claim is true for P_1 and P_2. Then it is true for $P = \texttt{if}\ e'\ \texttt{then}\ P_1\ \texttt{else}\ P_2$, in both cases of linear and non-linear conditionals. In particular when x is e', e must have type bool, because it has the same type as x. Then the substitution applies.

Again, assume that the claim is true for $\Gamma_1 \vdash P_1$ and for $\Gamma_2 \vdash P_2$. Let Γ_1' be the typing of $P_1\{e/x\}$ and let Γ_2' be the typing of $P_2\{e/x\}$. Note that $\Gamma_1'(e) = \Gamma_1(x)$ and $\Gamma_2'(e) = \Gamma_2(x)$. Then $\Gamma_1'(e) \asymp \Gamma_2'(e)$ and $\Gamma_1'(e) \odot \Gamma_2'(e) = \Gamma_1(x) \odot \Gamma_2(x)$. Then the claim is true for $P_1 \mid P_2$. □

Lemma E.3 (weakening). *Let $\Gamma \vdash P$, where P is not a selection. For any name u which does not appear in P we have $\Gamma, u : \bot \vdash P$ and $\Gamma, u : \oplus^\star_{i\in I} l_i(\vec{\tau}_i) \vdash P$.*

Proof. The claim is true for $\Gamma \vdash \mathbf{0}$. Then for every other typing rule we just assume that it is true for the premise(s) of the rule and by adding the typing $u : \bot$ or $u : \oplus^\star_{i\in I} l_i(\vec{\tau}_i) \vdash P$ to the premises, we also have it in the conclusion. Note in particular that when we impose constraints on the modalities in the typings of many premises, these do not apply to $\bot$

as well as to other base types which may not have modalities, nor do they prevent the non-linear output modality.

There are two specific cases which deserve further comments. First, in the rule for parallel composition, the weakening is applied only to one premise. Second, in the rule for output note that the premise may only type the names which appear in $\vec{e}$. Non-linear output names, as well as those with type $\bot$, can be added to the conclusion. □

Now we proceed to:

Proof of Proposition E.1 (1) (subject reduction): We consider the two reduction rules, showing they yield typable processes in all contexts.

Let $P \equiv \texttt{if}\ e\ \texttt{then}\ P'\ \texttt{else}\ P''$. By the typing rule for the conditional, e is a boolean expression. Then either $e \downarrow \texttt{tt}$ or $e \downarrow \texttt{ff}$. As a result, either $P \longrightarrow P'$ or $P \longrightarrow P''$. In any case, both $\Gamma \vdash P'$ and $\Gamma \vdash P''$ come as premises of the typing rule for the conditional.

Now let $P \equiv u\&_{i\in I}\{l_i(\vec{x}_i).P_i\} \mid \overline{u}\oplus l_j\langle\vec{e}\rangle$. By the typing rule for parallel composition, the typings on the two sides must assign dual types to u. Let $\Gamma_1, u\!:\!\oplus^{\mathrm{M}}_{i\in I} l_i(\vec{\tau}_i) \vdash \overline{u}\oplus^{\mathrm{M}} l_j\langle\vec{e}\rangle$ and $\Gamma_2, u\!:\!\&^{\mathrm{M}}_{i\in I} l_i(\overline{\vec{\tau}_i}) \vdash u\&^{\mathrm{M}}_{i\in I} l_i(\vec{x}_i).P_i$, where $\Gamma_1 \asymp \Gamma_2$ and $\overline{\vec{\tau}}$ is the vector of the duals of the projections of $\vec{\tau}$ (in the same order and for any type τ). Assuming that $\vec{e}\downarrow\vec{v}$, we have $P \longrightarrow P_j\{\vec{v}/\vec{x}_j\}$ and we need to show that:

$$\Gamma = (\Gamma_1, u\!:\!\oplus^{\mathrm{M}}_{i\in I} l_i(\vec{\tau}_i)) \odot (\Gamma_2, u\!:\!\&^{\mathrm{M}}_{i\in I} l_i(\overline{\vec{\tau}_i})) \vdash P_j\{\vec{v}/\vec{x}_j\}$$

We do this by cases of M. First let M be L. The typing rule for linear branching implies $\Gamma_1, \vec{x}_j : \vec{\tau}_j \vdash P_j$, while the typing rule for selection implies $\Gamma_2 \vdash \vec{e} : \vec{\tau}_j$ where the domain of Γ_2 contains all and only the names which appear in $\vec{e}$. Then by Lemma E.2, $\Gamma_1 \odot \Gamma_2 \vdash P_j\{\vec{v}/\vec{x}_j\}$. Note that, in addition to $\Gamma_1 \odot \Gamma_2$, Γ also contains $u : \oplus^{\mathrm{M}}_{i\in I} l_i(\vec{\tau}_i) \odot \&^{\mathrm{M}}_{i\in I} l_i(\overline{\vec{\tau}_i})$, which becomes $u : \bot$ since $\mathrm{M} = L$. Since u does not appear anywhere in $P_j\{\vec{v}/\vec{x}_j\}$, we can apply Lemma E.3 and state $\Gamma \vdash P_j\{\vec{v}/\vec{x}_j\}$.

When the modality is non-linear the case is similar, except that the input type does not disappear after the composition. And (accordingly) the typing rule for the branching has the typing for u also in the premise.

Let P be of the form $(\nu a)P'$ and $P' \longrightarrow P''$. By induction hypothesis, P'' can be typed with the same type as P'. Then $\Gamma \vdash (\nu a)P''$. Similarly for the case of the parallel composition. □

Proposition E.1 (2, 3) (Linear Actions). (2) (Active modes). Linear inputs cannot be suppressed by any other input, nor by a non-linear conditional, as imposed by the typing rules. Then they will become active after a sequence of reductions of linear conditionals. The same also applies to linear outputs, provided that the output name c does not occur as object in the process P.

(3) (Partial confluence). First let $P \longrightarrow P_1$ be the reduction of a linear conditional. Such a reduction does not exclude any other reduction because it only requires the evaluation of the expression in its condition and the decision of which branch to take. Then there is R such that $P_1 \longrightarrow R$, which reduces the same redex as $P \longrightarrow P_2$. For the same reason, $P \longrightarrow P_1$ may not be excluded by other reductions either. Then $P_2 \longrightarrow R$, which reduces the same redex as $P \longrightarrow P_1$.

Now let $P \longrightarrow P_1$ be the reduction of a synchronisation between a linear branching and its complementary selection. Note that: 1) a linear branching may only appear once because its subject name does not appear in the premise of the typing rule which introduces it; 2) a linear selection may only appear once because it may not be composed with another selection (by definition of $\asymp$) and because once it is composed with a complementary branching, the type of its subject becomes $\perp$ which may not be composed anymore. Then the reduction $P \longrightarrow P_1$ does not exclude any other reduction and may not be excluded by any other reduction. Then the claim holds just as it did in the case of the conditional. □

E.2 Subject Transition

The following is a basic property of the typed LTS.

Proposition E.4 (subject transition). *If $\Gamma \vdash P$, $P \xrightarrow{\ell} Q$ and $\Gamma \xrightarrow{\ell} \Delta$ then $\Delta \vdash Q$.*

Proof. The case of τ-action is subsumed by Proposition E.1 (1) (subject reduction). Thus the only cases of interest are the input and output visible transitions.

(L-input). The environment transition reads:

$$\Gamma,\, a : \&^{L}\{l_i(\vec{\tau}_i)\}_{i\in I} \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\&l_j\langle\vec{v}\rangle} \Gamma,\, \vec{v} : \vec{\tau}_j,\, a : \perp$$

Note in particular that the names in $\vec{v}$ are not in the domain of Γ, since they are fresh. Now, the process transition adds $a \oplus l_j\langle\vec{v}\rangle$ as a parallel component (and no change otherwise). We can infer the type of $a \oplus l_j\langle\vec{v}\rangle$ as:

$$\vec{v} : \vec{\tau}_j, a : \oplus^{L}\{l_i(\overline{\vec{\tau}_i})\}_{i\in I} \vdash a \oplus l_j\langle\vec{v}\rangle$$

Then we just need to apply the (Par) rule in § C.1 and we are done.

($\star$-input). The process transition is the same as above, only the environment transition changes slightly, as follows:

$$\Gamma,\, a : \&^{\star}\{l_i(\vec{\tau}_i)\}_{i\in I} \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\&l_j\langle\vec{v}\rangle} \Gamma \odot \vec{v} : \overline{\vec{\tau}_j},\, a : \&^{\star}\{l_i(\vec{\tau}_i)\}_{i\in I}$$

reflecting the fact that the type composition, $\odot$, between non-linear types is defined to keep the input type, rather than $\perp$. The rest follows the same reasoning as above.

(L-output. The environment transition reads:

$$(\Gamma \odot \vec{v} : \overline{\vec{\tau}_j})/\vec{c},\, a : \oplus^{L}\{l_i(\vec{\tau}_i)\}_{i\in I} \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l_j\langle\vec{v}_j\rangle} \Gamma$$

where $\vec{v} \subseteq Dom(\Gamma)$ (so that linear names are typed $\perp$ in the left-hand environment). At the term level, this corresponds to:

$$(\boldsymbol{\nu}\,\vec{c})(P|\overline{a} \oplus l\langle\vec{v}\rangle) \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l\langle\vec{v}\rangle} P$$

Note that $\vec{c}$ must be empty because only non-linear names or names typed $\perp$ may be hidden, but linear inputs may only carry linear output names (which are not typed with $\perp$) or constants. Thus we infer:

$$\Gamma \odot \vec{v} : \overline{\vec{\tau}_j},\, a : \oplus^{L}\{l_i(\vec{\tau}_i)\}_{i\in I} \quad \vdash \quad P|\overline{a} \oplus l\langle\vec{v}\rangle$$

from which we can infer:

$$\Gamma \quad \vdash \quad P$$

as required.

($\star$-output. The process transition is the same as above, while the environment transition changes slightly, as follows:

$$(\Gamma \odot \vec{v} : \overline{\vec{\tau}_j})/\vec{c},\, a : \oplus^\star\{l_i(\vec{\tau}_i)\}_{i\in I} \xrightarrow{(\boldsymbol{\nu}\,\vec{c})a\oplus l_j\langle\vec{v}\rangle} \Gamma,\, a : \oplus^\star\{l_i(\vec{\tau}_i)\}_{i\in I}$$

where $\vec{v} \subseteq Dom(\Gamma)$ still holds. Although $\vec{c}$ may be non-empty, the typing and input/output alternation still allow us to infer:

$$\Gamma \odot \vec{v} : \overline{\vec{\tau}_j},\, a : \oplus^\star\{l_i(\vec{\tau}_i)\}_{i\in I} \quad \vdash \quad P|\overline{a} \oplus l\langle\vec{v}\rangle$$

from which we can infer that either:

$$\Gamma,\, a : \oplus^\star\{l_i(\vec{\tau}_i)\}_{i\in I} \quad \vdash \quad P$$

or $\Gamma \vdash P$. But on the latter we can still apply Lemma E.3 and obtain the former again. □

E.3 Proposition 2.3 (Linear Actions)

Proof of Proposition 2.3 (1) is standard. (2) is from (1) and typing rules. (3) is direct from the definitions (note that, by construction, a typed transition sequence always obeys the standard binder convention). □

E.4 Proposition 2.5 (congruence of $\approx$)

In this section, we use the following notion of context:

$$\begin{array}{rcl} C[-] & ::= & u\&_{i\in I\setminus\{j\}}\{l_i(\vec{x}_i).P_i\}\&\{l_j(\vec{x}_j).C[-]\} \;\mid\; C[-]|P \;\mid\; P|C[-] \;\mid\; (\boldsymbol{\nu}\,u)C[-] \\ & \mid & \texttt{if}\ e\ \texttt{then}\ C[-]\ \texttt{else}\ P \;\mid\; \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ C[-] \;\mid\; - \end{array}$$

The following lemma is used in the congruence proof to deal with recursion.

Lemma E.5. $C_1[X] \approx C_2[X] \wedge P_1 \approx P_2 \Rightarrow C_1[P_1] \approx C_2[P_2]$

Proof. Let $R = \{(P_1, P_2) \mid P_1 = C_1[Q_1] \wedge P_2 = C_2[Q_2] \wedge C_1[X] \approx C_2[X] \wedge Q_1 \approx Q_2\}$. We need to show that R is a bisimulation.

Let $P_1 = C_1[Q_1]\ R\ C_2[Q_2] = P_2$, where $C_1[X] \approx C_2[X]$ and $Q_1 \approx Q_2$. Let also $P_1 \xrightarrow{\ell} P_1'$. We only show the case where both the context C_1 and the sub-process Q_1 are reduced, since the other cases are included in this one. Then $\ell = \tau$ and the transition $P_1 \xrightarrow{\tau} P_1'$ results from a synchronisation between a transition of C_1 and the dual transition of P_1. Let them be $C_1[X] \xrightarrow{\ell'} C_1'[X]$ and $Q_1 \xrightarrow{\ell''} Q_1'$ respectively, where ℓ' is the dual of ℓ'' and $P_1' = C_1'[Q_1']$. By bisimilarity, we have $C_2[X] \xrightarrow{\ell'} C_2'[X]$ and $Q_2 \xrightarrow{\ell''} Q_2'$, where $C_1'[X] \approx C_2'[X]$ and $Q_1' \approx Q_2'$. Then we define $P_2' = C_2'[Q_2']$ and conclude $P_1'\ R\ P_2'$. □

We are now ready to prove congruence. We also recall the definition given in § 3 of Γ *allows* ℓ, written $\Gamma \vdash \ell$, that is when $\Gamma \xrightarrow{\ell} \Gamma'$ for some Γ'.

Proof of Proposition 2.5 We prove the two points separately:

1. We give the congruence proof for the main two cases: parallel composition and recursion. The other cases are standard.
   First let:

$$R = \{(\Gamma \vdash P_1, \Gamma \vdash P_2) \mid P_1 \equiv (\nu\vec{u})(P_1' \mid R) \wedge P_2 \equiv (\nu\vec{u})(P_2' \mid R) \wedge \Gamma' \vdash P_1' \approx P_2'\}$$

   We need to show that R is a bisimulation. Then let $\Gamma \vdash P_1 \xrightarrow{\ell} \Delta \vdash Q_1$ and $\Gamma \vdash P_2 \xrightarrow{\ell} \Delta \vdash Q_2$, where $\Gamma \vdash P_1 \; R \; P_2$. If ℓ is an input label, the two transitions above add the same output action in parallel to the processes P_1 and P_2, respectively. Then $(Q_1, Q_2) \in R$, by the definition of R itself.
   Now let ℓ be an output label and let $P_1 \equiv (\nu\vec{u})(P_1' \mid R)$ and $P_2 \equiv (\nu\vec{u})(P_2' \mid R)$, where $P_1' \approx P_2'$. Note that the subject of ℓ may not be in $\vec{u}$, else the output would not be allowed by Γ. If the output comes from R, then it removes an output action from R. Then again $(Q_1, Q_2) \in R$ by definition, as in the input case. Otherwise, it comes from P_1' and P_2': $P_1' \xrightarrow{\ell} Q_1'$ and $P_2' \xrightarrow{\ell} Q_2'$, where $Q_1 \equiv (\nu\vec{u})(Q_1' \mid R)$ and $Q_2 \equiv (\nu\vec{u})(Q_2' \mid R)$. Since $Q_1' \approx Q_2'$, we conclude $(Q_1, Q_2) \in R$.
   Let $\ell = \tau$. If the transition comes from either side of the parallel composition, the same reasoning we gave for the output case applies. Then let it be a synchronisation between the two sides. The argument is again similar: on one side the two transitions reduce the same action in R, reaching R' in both cases; on the other side P_1' and P_2' are reduced to Q_1' and Q_2', respectively, such that $Q_1' \approx Q_2'$. As a whole, we obtain $Q_1 \equiv (\nu\vec{u})(Q_1' \mid R')$ and $Q_2 \equiv (\nu\vec{u})(Q_2' \mid R')$ and we have $(Q_1, Q_2) \in R$, by definition.
   The other case we consider is recursion. We define (omitting the typings):

$$R = \{(P_1, P_2) \mid P_1 = C_1[(\mu X(\vec{x}).P_1')\langle\vec{e}\rangle] \wedge P_2 = C_2[(\mu X(\vec{x}).P_2')\langle\vec{e}\rangle] \wedge P_1'\{\vec{e}/\vec{x}\} \approx P_2'\{\vec{e}/\vec{x}\} \wedge C_1[X] \approx C_2[X]\}$$

   Again we need to show that R is a bisimulation. We only do one direction, the other is symmetric. Then let $P_1 \; R \; P_2$ and $P_1 \xrightarrow{\ell} Q_1$. First suppose that the transition above reduces the context of P_1, i.e. $P_1 = C_1[(\mu X(\vec{x}).P_1')\langle\vec{e}\rangle] \xrightarrow{\ell} C_1'[(\mu X(\vec{x}).P_1')\langle\vec{e}\rangle] = Q_1$. Since only the context is affected, we also have $C_1[X] \xrightarrow{\ell} C_1'[X]$. By definition of R, $P_2 = C_2[(\mu X(\vec{x}).P_2')\langle\vec{e}\rangle]$, for some context $C_2[-]$ such that $C_1[X] \approx C_2[X]$. Then $C_2[X] \xrightarrow{\ell} C_2'[X]$ and $C_1'[X] \approx C_2'[X]$. Then we also have $P_2 \xrightarrow{\ell} C_2'[(\mu X(\vec{x}).P_2')\langle\vec{e}\rangle] = Q_2$ and $Q_1 \; R \; Q_2$.
   Now let $P_1 = C_1[(\mu X(\vec{x}).P_1')\langle\vec{e}\rangle] \equiv C_1[P_1'\{\vec{e}/\vec{x}\}\{\mu X(\vec{x}).P_1'/X\}]$ and $P_1 \xrightarrow{\ell} C_1[P_1''\{\vec{e}/\vec{x}\}\{\mu X(\vec{x}).P_1'/X\}] = Q_1$. By definition of R, $P_2 = C_2[(\mu X(\vec{x}).P_2')\langle\vec{e}\rangle] \equiv C_2[P_2'\{\vec{e}/\vec{x}\}\{\mu X(\vec{x}).P_2'/X\}]$, where $P_1'\{\vec{e}/\vec{x}\} \approx P_2'\{\vec{e}/\vec{x}\}$ and $C_1[X] \approx C_2[X]$. Let $C_1'[-] = C_1[P_1'\{\vec{e}/\vec{x}\}\{-/X\}]$ and $C_2'[-] = C_2[P_2'\{\vec{e}/\vec{x}\}\{-/X\}]$ and note that $C_1'[X] \approx C_2'[X]$, by Lemma E.5. Then $P_2 \xrightarrow{\ell} C_2[P_2''\{\vec{e}/\vec{x}\}\{\mu X(\vec{x}).P_2'/X\}] \equiv Q_2$. Now let $C_1''[-] = C_1[P_1''\{\vec{e}/\vec{x}\}\{-/X\}]$ and $C_2''[-] = C_2[P_2''\{\vec{e}/\vec{x}\}\{-/X\}]$, which are such that $C_1''[X] \approx C_2''[X]$. Then $Q_1 \; R \; Q_2$, by definition of R.
   The final case, where both the context and the recursive sub-process inside the context are reduced uses Lemma E.5 again, and is a combination of the above two cases.

2. Let $R = \{(\Gamma \vdash P, \Gamma \vdash Q) \mid P \longrightarrow_{\mathsf{L}} Q \vee P = Q\}$. Note that $\longrightarrow_{\mathsf{L}} \subset R$, then we need to show that R is a bisimulation. Since the identity is clearly a bisimulation, let $P \longrightarrow_{\mathsf{L}} P'$. If $P' \xrightarrow{\ell} Q'$ then $P \longrightarrow_{\mathsf{L}} \xrightarrow{\ell} Q'$ and $(Q', Q') \in R$. If $P \xrightarrow{\ell} Q$, this transition may either coincide with $P \longrightarrow_{\mathsf{L}} P'$ or not. In the former case, $P' = Q$ then $(P', Q) \in R$. In the latter case, note that none of the actions reduced by $P \longrightarrow_{\mathsf{L}} P'$ may be reduced by $P \xrightarrow{\ell} Q$, because they are linear actions and may only occur once. Then $P' \xrightarrow{\ell} Q'$ and $(Q, Q') \in R$. □

F Proofs for Section 3

F.1 Proposition 3.14 (relationship among progress properties)

In the proof of Proposition 3.14, the least trivial inclusion to show is that weak non-blockingness implies reliability. The proof relies on the following lemma.

Lemma F.1. *Let $\Gamma \vdash P$. Then there exists a non-failing fair sequence of transitions from $\Gamma \vdash P$ which does not contain any non-linear input.*

Proof. Since we do not have to perform a non-linear input in a fair transition sequence, we can simply construct, for example by a round-robin scheduling, a fair transition sequence from P, so that we perform all transitions one by one *except* non-linear inputs. One concrete strategy is given in §I.1. □

Now we can show that weak non-blockingness implies reliability.

Proposition F.2. *Let $\Gamma \vdash P$ be weakly non-blocking. Then $\Gamma \vdash P$ is reliable.*

Proof. Consider an arbitrary non-failing transition sequence from $\Gamma \vdash P$:

$$\Psi' : \Gamma \vdash P \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_n} \Delta \vdash Q$$

such that $c \in \mathtt{allowed}(\Delta)$. By Lemma F.1, there exists a further non-failing and fair transition sequence, which does not contain any non-linear input:

$$\Psi'' : \Delta \vdash Q \xrightarrow{\ell_{n+1}} \ldots$$

where Ψ'' may be either finite or infinite. Let Ψ be the concatenation of Ψ' and Ψ''. Since $c \in \mathtt{allowed}(\Delta)$ and Ψ is non-failing, we have $\mathtt{allowed}(\Delta) \setminus \mathtt{blocked}(\Psi) = \mathtt{allowed}(\Delta) \neq \emptyset$.

Since $\Gamma \vdash P$ is weakly non-blocking, since Ψ is fair (hence also maximal) and since $\mathtt{allowed}(\Delta) \setminus \mathtt{blocked}(\Psi) \neq \emptyset$, some output must occur in Ψ'':

$$\Psi'' : \Delta \vdash Q \xrightarrow{\ell_{n+1}} \ldots \xrightarrow{(\boldsymbol{\nu}\, \vec{a^1}) c_1 \oplus l\langle \vec{v^1} \rangle} \Delta' \vdash Q' \xrightarrow{\ell'_1} \ldots$$

If $c_1 = c$ we can immediately conclude that $\Gamma \vdash P$ is reliable. Then we need to consider the case where $c_1 \neq c$. The latter implies that $c \in \mathtt{allowed}(\Delta')$, which in turn implies that $\mathtt{allowed}(\Delta') \setminus \mathtt{blocked}(\Psi) \neq \emptyset$. Then we can apply the above reasoning again, stating that some output occurs in Ψ'' after $\Delta' \vdash Q'$:

$$\Psi : \Delta \vdash Q \xrightarrow{\ell_{n+1}} \ldots \xrightarrow{(\boldsymbol{\nu}\, \vec{a^1}) c_1 \oplus l\langle \vec{v^1} \rangle} \Delta' \vdash Q' \xrightarrow{\ell'_1} \ldots \xrightarrow{(\boldsymbol{\nu}\, \vec{a^2}) c_2 \oplus l\langle \vec{v^2} \rangle} \Delta'' \vdash Q'' \xrightarrow{\ell''_1} \ldots$$

Note that $\texttt{allowed}(\Delta)$ is finite. Moreover, since no non-linear input occurs in Ψ'', $\texttt{allowed}(\Delta'')$ is strictly contained in $\texttt{allowed}(\Delta')$, which is strictly contained in $\texttt{allowed}(\Delta)$. Then we are ensured that an output on c will occur in a finite number of steps.

Then $\Gamma \vdash P$ is reliable. □

Finally we can prove the inclusions between all the behavioural properties and the preservation of the two global progress properties by the fair pre-order.

We now move to:

Proof of Proposition 3.14:
1) Trivial.
2) First we show the inclusions.

1. **WNB implies RLB.** By Lemma F.2, if $\Gamma \vdash P$ is weakly non-blocking it is also reliable.
2. **NB implies WNB.** If $\Gamma \vdash P$ is non-blocking it is also weakly non-blocking because its requirement holds for all derivatives, which may be obtained by failing or by non-failing transition sequences.
3. **WF implies WWF.** Same as the above.
4. **WF implies NB.** If $\Gamma \vdash P$ is wait-free it is also non-blocking, because where wait-freedom asks for a specific output, non-blockingness asks for any output.
5. **WWF implies WNB.** Same as the above.

The examples introduced in § 4 are enough for the strictness constraints. Below we give a few more examples, where the basic properties are distilled from the rest.

1. **RLB but not WNB.** We can take:

$$\begin{array}{l}\mathsf{Ref}\langle r, \mathsf{ff}\rangle \mid \mu X().r \triangleleft \mathsf{tt}\langle.\rangle r \triangleleft \mathsf{ff}\langle.\rangle X\langle\rangle \mid \\ \mu X().r \triangleleft read(x).\texttt{if } x \texttt{ then } \bar{y} \texttt{ else } X\langle\rangle\end{array}$$

   where

$$x \triangleleft v\langle.\rangle P \stackrel{\text{def}}{=} (\nu c)(\overline{x} \oplus write\langle v, c\rangle | c().P)$$

   While one sub-process keeps changing the value of the reference r from ff to $\mathtt{tt}$, the other keeps checking that this value is $\mathtt{tt}$ before it can output. First of all, it is understood that any typing of this process must allow an output on y. Furthermore, any derivative admits a transition sequence where the output occurs, then the process is reliable. However, it also admits a fair sequence where it does not happen (independently of failures). Then it is not weakly non-blocking.
2. **NB but not WWF.** We can take:

$$\begin{array}{l}\mathsf{Ref}^{\mathsf{cas}}\langle r, 0\rangle \mid !u(y).\mu X.r \triangleleft read(x). \\ \qquad\qquad \texttt{if } \texttt{cas}(r, x, x+1) \texttt{ then } \bar{y} \texttt{ else } X\langle\rangle\end{array}$$

   Here, potentially many sub-processes may be spawned from the one to the right of the parallel sign. Each one of them will try to increment the value contained in the reference r, and then output. But they can only do it if they were not preceded by others, else they keep looping. Note also that the failure of one sub-process may not block the others[3]. Then as long as there are allowed outputs that are not blocked,

[3] The detailed proof follows the same reasoning given for $\mathsf{CQemp}(r)$ (§ 4).

some output will occur in any fair non-failing sequence. Then the above process is non-blocking. On the other hand, some allowed output may never happen (associated sub-process loops forever). Then it is not weakly wait-free.

3. **WWF but not WF.** We can take:

$$\mathsf{Mtx}\langle l\rangle|!u(y).\bar{l}(g)g(z).\bar{z} \mid \bar{y}$$

where $\bar{z}$ may fail because it is non-linear, thus blocking all other sub-processes which may be waiting to synchronise with l. □

G Proofs for Section 4

G.1 Molecular Actions

We start by restating formally the notion of molecular action which was introduced in § 4.

Definition G.1 (molecular action). A typed process $\Gamma \vdash P$ is in *linear normal form*, or lin-nf, when P does not admit any linear redex, $P \not\longrightarrow_{\mathsf{L}}$. If P is in lin-nf, we write $\Gamma \vdash P \overset{\ell}{\mapsto} \Delta \vdash Q$ when, for some R, we have $\Gamma \vdash P \overset{\ell}{\to} \Delta \vdash R$ and $R \overset{\tau}{\to}_{\mathsf{L}}^{*} Q \not\longrightarrow_{\mathsf{L}}$. We call the transition $\overset{\ell}{\mapsto}$, a *molecular action*.

Observe that, by Prop. 2.3 (2), any typed P can reduce to a unique lin-nf up to $\equiv$.

Proposition G.2 (molecular action). *If $\Gamma \vdash P \overset{\ell}{\to} \Delta \vdash Q$, then $\Gamma \vdash P \overset{\ell}{\mapsto} \Delta \vdash Q'$ such that $Q \overset{\tau}{\to}_{\mathsf{L}}^{*} Q'$.*

Proof. Immediate from Prop. 2.3 (1) and (2). □

Proposition G.3. *If $\mathcal{R}$ is a bisimulation over lin-nfs using molecular actions as one-step transitions, and if $P\mathcal{R}Q$, then $P \approx Q$.*

Proof. If the condition holds then $\overset{\tau}{\to}_{\mathsf{L}}^{*} \circ \mathcal{R} \circ (\overset{\tau}{\to}_{\mathsf{L}}^{*})^{-1}$ is a bisimulation. □

The following convention and proposition formalise what we stipulated in § 4.3 that we only consider *molecular action derivatives* of $\mathsf{CQemp}(r)$, called cas-queue processes.

Convention G.4 (cas-queue process). *Henceforth, a* cas-queue process *always denotes a derivative of a typed transition sequence from $\mathsf{CQemp}(r)$ that is also in lin-nf.*

Proposition G.5. *A* cas*-queue process under Convention G.4 can always be reached from the empty queue by a sequence of zero or more molecular actions.*

Proof. By Proposition G.2 and noting $\mathsf{CQemp}(r)$ is in lin-nf by construction. □

G.2 Shape of cas-Queue Processes

We start by better characterising the nature of threads which, as we know, are all in local molecular form (Definition 4.2).

Proposition G.6. *Each LMF contains a unique active output.*

Proof. Immediate by inspection. □

The g-thread in a cas-queue process P is sequential, in the following sense. Below we say a thread in a cas-queue process is *involved in* a τ-action when a sub-term of the thread participates in the reduction (either as an active input, an active output, or a conditional). Similarly we say a thread is *involved in a molecular τ-action* if a sub-term of the thread participates in the first reduction of the molecular action.

Proposition G.7 (single activity). *Assume a g_i-thread in a* cas*-queue process P is involved in a molecular τ-action. Then this molecular action is determined uniquely. Moreover it has a molecular τ-action iff the g_i-thread is not an answer.*

Proof. By Lemma G.6, the initial redex is unique. The rest is by inspection of the molecular action associated to the initial redex in each LMF. Note that such a molecular action is uniquely identified by Proposition G.2. □

Next, we formalise the classification of molecular actions we gave in § 4.3.

Definition G.8. Hereafter we write $P \overset{\boldsymbol{\tau},g}{\longmapsto} Q$ when, for any cas-queue process P, there is a molecular τ-action $P \overset{\tau}{\longmapsto} Q$ in which the g-thread of P is involved.

Definition G.9 (LMF actions). Let P be a cas-queue process.

1. We write $P \overset{\mathsf{com}(g)}{\longmapsto} Q$ when $P \overset{\boldsymbol{\tau},g}{\longmapsto} Q$ where the first τ transition is a *commit*. We call this action *commit action*.
2. We write $P \overset{\mathsf{sw}(g)}{\longmapsto} Q$ when $P \overset{\boldsymbol{\tau},g}{\longmapsto} Q$ which is a tail-swinging cas-operation, i.e. the initial reduction is from one of the following:
   (a) $\mathsf{EnqSwRec}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle$,
   (b) $\mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle$ and
   (c) $\mathsf{DeqSw}_{\mathsf{cas}}\langle nxt, t\_ctr, tn, h, t, g\rangle$).

   We call this action *swinging action*. Note a swinging action may not affect the state when the swinging is already done. We may write $\overset{\mathsf{sw}(g_1..g_n)}{\longmapsto}$ for a series of (successful and failing) swinging actions at $g_1..g_n$.
3. We write $P \overset{\mathsf{nc}(g)}{\longmapsto} Q$ for any $P \overset{\boldsymbol{\tau},g}{\longmapsto} Q$ that is not a commit action. We call it a *non-commit action*.

Note: 1) a swinging action is a non-commit action, but it is worth treating it separately since it may modify the queue non-trivially; 2) the reduction of a given LMF may be a commit action in certain contexts and not in others.

Definition G.10 (ready output). An output in P is *ready* if either it is the output part of a redex pair in P or it is ready to induce a typed output transition.

The following result was used in the proof of Lemma 4.6.

Proposition G.11. *Let $\Gamma \vdash P$ be a* cas*-queue process in general form.*

1. *Exactly one output in each thread of P is ready in P, and these outputs cover all the outputs that are ready in P.*
2. *If $\Gamma \vdash P \xrightarrow{\ell} \Gamma' \vdash P'$ and ℓ is an input or an output, then P' is still in general form.*

Proof. (1) is immediate. For (2), for input, the threads ΠP_i are combined with $R \overset{\text{def}}{=} \overline{r} \oplus \mathtt{enq}\langle v, g\rangle$. By the use of typed transition, g is fresh. The linked list sub-process does not change. The additional R is ready to interact at r, hence done. The output case is simpler since we only lose one message (one thread is removed). □

G.3 Normal Form and Normalisation

In this section we prove some partial results which are used in the proof of Lemma 4.11.

Proposition G.12 (commit and swing). *Let $\Phi : \mathsf{CQemp}(r) \overset{\tau}{\longmapsto}{}^{*} P$. Then Φ contains at most one commit at g. Moreover, if it does, then Φ contains at most one swinging molecular action in the g-thread after the commit.*

Proof. The proof is done by inspection of the LMFs we may reach after a commit. It may have one of the following three forms: 1) $\mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle$ (after a commit in an enqueue operation); 2) $\mathsf{DeqAnsNull}_{\mathsf{cas}}\langle g\rangle$ (after a commit in a dequeue operation where the queue was empty); 3) $\mathsf{DeqAns}_{\mathsf{cas}}\langle v, g\rangle$ (after a commit in a dequeue operation where a value has been dequeued). In the latter two cases, we have already an answer which has no further redex. In the former case, we need a further molecular action and then we reach an answer. □

Note also that a swinging action in one thread never disables a commit in another.

The following result provides the base case of the normalisation lemma.

Proposition G.13 (base normalisation). *Let P be an initial* cas*-queue process, $\{g_1, ..., g_m\}$ exhaust the thread names in P, and $K_1 \cdot K_2$ be a non-redundant sequence of these thread names. Then $P \xmapsto{\mathsf{norm}(K_1 \cdot K_2)} Q$ such that the local normalisations in the partition K_1 are committing and those in K_2 are pre-committing.*

Proof. We show the local normalisation of an arbitrary g-thread, which may be either of the form $\mathsf{EnqReq}_{\mathsf{cas}}\langle r, v, g\rangle$, or of the form $\mathsf{DeqReq}_{\mathsf{cas}}\langle r, g\rangle$, since P is initial. The following invariant ensures that such normalisations can be applied one after the other:

> *"After each local normalisation sequence, we reach a general form where the tail pointer points to the last node in the linked list sub-process."*

$\mathsf{EnqReq}_{\mathsf{cas}}\langle r, v, g\rangle$ synchronises with $\mathsf{CQ}(r, h, t)$, spawning the thread $\mathsf{EnqRdT}_{\mathsf{cas}}\langle t, v, g\rangle$. The next molecular actions perform read operations on: the tail pointer, the node it refers to (which is the last node by the invariant) and the pointer to its successor. Since the successor of the last node is $null$, we reach $\mathsf{EnqCom}_{\mathsf{cas}}\langle nxt, tp\_ctr, tp, tn, t\_ctr, t, v, g\rangle$ which is a pending local normal form, since the tail pointer points to the last node.

Now, if we need a non-committing normalisation (g is in the K_2 partition) we are done. Otherwise we commit and we move to $\mathsf{EnqSwFin}_{\mathsf{cas}}\langle tn, t\_ctr, t, nd, g\rangle$, which swings the tail forward (preserving the invariant). Then we reach the answer $\mathsf{EnqAns}_{\mathsf{cas}}\langle g\rangle$, which completes the local normalisation.

Similarly, $\mathsf{DeqReq}_{\mathsf{cas}}\langle r, g\rangle$ synchronises with $\mathsf{CQ}(r, h, t)$, spawning a thread of the form $\mathsf{DeqRdH}_{\mathsf{cas}}\langle h, t, g\rangle$. Again we perform the required read operations and we reach $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt, hp\_ctr, tn, t\_ctr, hn, h\_ctr, h, t, g\rangle$. Now, if h and t point to the same node, it must be the last node in the linked list, because of the invariant. Then we are at a pending local normal form, which is enough if g is in the K_2 partition. Then let g be in the K_1 partition: the next two conditionals (line 6 and 7 in the definition of the dequeue operation, § D) evaluate to $true$, because h and t point to the same node and because the successor of this node is $null$, since it is the last. Then we reach the answer $\mathsf{DeqAnsNull}_{\mathsf{cas}}\langle g\rangle$, preserving the invariant and completing the local normalisation.

Now assume that in $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle nxt, hp\_ctr, tn, t\_ctr, hn, h\_ctr, h, t, g\rangle$, h and t do not point to the same node. Then the condition at line 6 evaluates to $false$ and we reach $\mathsf{DeqRdNext}_{\mathsf{cas}}\langle h\_ctr, hn, h, g\rangle$, and then $\mathsf{DeqCom}_{\mathsf{cas}}\langle x, t, h\_ctr, hn, h, g\rangle$, which is a pending local normal form satisfying the invariant. This is enough if g is in the K_2 partition. Otherwise, we perform the cas which swings h forward, reaching the answer $\mathsf{DeqAns}_{\mathsf{cas}}\langle v, g\rangle$, and we are done. □

The proof of the normalisation lemma relies on local permutations, the main permutation cases were stated but not proved (Proposition 4.10). Here we show them along with further auxiliary cases.

Lemma G.14 (local permutation (sw-sw)). *Let P be a cas-queue process. $P \xmapsto{\mathsf{sw}(g_i)} \xmapsto{\mathsf{sw}(g_j)} R$ implies $P \xmapsto{\mathsf{sw}(g_j)} \xmapsto{\mathsf{sw}(g_i)} R$.*

Proof. Note that the transitions $P \xmapsto{\mathsf{sw}(g_i)} \xmapsto{\mathsf{sw}(g_j)} R$ correspond to two successive cas operations on the tail pointer. Then i and j must be different. Let $\langle nd_i, n_i\rangle$ and $\langle nd'_i, n'_i\rangle$ be, respectively, the old and the new value parameters of the first cas operation, the one in the g_i-thread. Similarly, let $\langle nd_j, n_j\rangle$ and $\langle nd'_j, n'_j\rangle$ be, respectively, the old and the new value parameters of the cas operation in the g_j-thread. Since they are both tail-swinging cas operations, we know that: 1) nd'_i is the successor of nd_i and nd'_j is the successor of nd_j in the linked list; 2) $n'_i = n_i + 1$ and $n'_j = n_j + 1$.

First suppose that $\langle nd_i, n_i\rangle \neq \langle nd_j, n_j\rangle$. Note that $\langle nd_i, n_i\rangle$ and $\langle nd_j, n_j\rangle$ are two previously read values of the tail pointer Then both $nd_i \neq nd_j$ and $n_i \neq n_j$, since each time the tail pointer is modified, both the node reference and the counter are modified. Then at least one of $\langle nd_i, n_i\rangle$ and $\langle nd_j, n_j\rangle$ must be outdated, since they are both previously read values of the tail pointer and they are different. Then at least one of the two tail-swinging operations would be ineffective at P. Then we can do the permutation without changing their outcome.

Now suppose that $\langle nd_i, n_i\rangle = \langle nd_j, n_j\rangle$. Then $n'_i = n_i + 1 = n_j + 1 = n'_j$, and also $nd'_i = nd'_j$, since they are both the successor of $nd_i = nd_j$ in the linked list. Now, surely the second operation does not affect the contents of the tail pointer. Hence, if the first operation does not modify the tail pointer either, the order is trivially irrelevant. Otherwise, when the first operation does modify the tail pointer, it sets $\langle nd'_i, n'_i\rangle$ as new value. Since $\langle nd'_i, n'_i\rangle = \langle nd'_j, n'_j\rangle$, we obtain the same transformation after the permutation. □

Below, we use the LMFs from Definition 4.4 to describe threads, whose reductions are described in Figures 1 (enqueue) and 2 (dequeue). Moreover, instead of explicitly writing the actual arguments of an LMF, we use generic substitutions $\phi, \phi', \ldots$, as in $\mathsf{EnqReq}_{\mathsf{cas}}\langle\phi\rangle$, $\mathsf{DeqRdNext}_{\mathsf{cas}}\langle\phi'\rangle, \ldots$. Finally, just as we defined a cas operation, we say that a $read$ *operation* is a molecular action in which a thread reads the content of a node or of a pointer.

The proof of Proposition 4.10 (nc-up) can be divided in two cases, according to whether the non-commit action to be moved left is a read operation or a tail-swinging cas. The following lemma deals with the former case.

Lemma G.15. *Let P be a queue process such that $P \xmapsto{\mathsf{norm}(g_j)} P' \xmapsto{\mathsf{nc}(g_i)} R$, where $P' \xmapsto{\mathsf{nc}(g_i)} R$ is a $read$ operation and $P \xmapsto{\mathsf{nc}(g_i)}$ (i.e. the same action is also a non-commit one at P).*

Then $P \xmapsto{\mathsf{nc}(g_i)} \xmapsto{\mathsf{norm}(g_j)} \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$ *and* $R \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$.

Proof. We need to show how to permute $P' \xmapsto{\mathsf{nc}(g_i)} R$ with any molecular action in $P \xmapsto{\mathsf{norm}(g_j)} P'$. Then assume that after an arbitrary number of permutations, we now need to permute our read operation with the molecular action $Q \xmapsto{\ell(g_j)}$:

$$Q \xmapsto{\ell(g_j)} \xmapsto{\mathsf{nc}(g_i)} \xmapsto{\mathsf{norm}(g_j)} Q'$$

where the final transition sequence is the rest of the normalisation for the g_j-thread, such that $Q' \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$ and $R \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$. Note also that $Q \xmapsto{\mathsf{nc}(g_i)}$, because the action $P \xmapsto{\mathsf{nc}(g_i)}$ could not be disabled by any action in the g_j-thread and because a non-commit action may not become a commit just by moving right (easy case analysis).

The only critical case is when $Q \xmapsto{\ell(g_j)}$ is a cas which modifies the contents of the reference, say x, which we are about to read. Note that in this case x must be a pointer, because we may not perform cas on a node. Since a read operation does not modify the linked list in any way, we can permute the read before the cas:

$$Q \xmapsto{\mathsf{nc}(g_i)} \xmapsto{\ell(g_j)} \xmapsto{\mathsf{norm}(g_j)} Q''$$

where the read operation is a non-commit action at Q as well, because the action $Q \xmapsto{\mathsf{nc}(g_i)}$ is unique (Prop. G.7). However, the value of x that is read is outdated in Q'' and needs to be read again. We show the claim by cases of x. In each case, the proof consists in making the g_i-thread of Q'' perform a complete loop to reach a process Q''' where the value of x is up-to-date. In some cases Q''' and Q' may not correspond exactly, because other references that were read before x may be outdated in Q'. In those cases we just need to apply the same reasoning described below to Q': we conclude that either $Q' \xmapsto{\mathsf{nc}(g_i)}{}^{*} Q''' \xmapsto{\mathsf{nc}(g_i)}{}^{*} R'$ or $Q' \xmapsto{\mathsf{nc}(g_i)}{}^{*} R' \xmapsto{\mathsf{nc}(g_i)}{}^{*} Q'''$. In both cases, the invariant that the final process (resp. R' or Q''') can be reached from R is still satisfied.

Hence, now we only need to show how to reach Q''' from Q''. The possible cases of x are: tail pointer, head pointer or last pointer in the linked list; since these are the only pointers which may be modified.

Let x be the tail pointer. Then $\ell(g_j)$ is actually a tail swinging operation:

$$Q \xmapsto{\mathsf{nc}(g_i)} \xmapsto{\mathsf{sw}(g_j)} \xmapsto{\mathsf{norm}(g_j)} Q''$$

Note that in Q'', the g_i-thread is either of the form $\mathsf{EnqRdTN}_{\mathsf{cas}}\langle\phi_e\rangle$ or of the form $\mathsf{DeqRdHN}_{\mathsf{cas}}\langle\phi_d\rangle$ (for proper substitutions ϕ_e and ϕ_d), as these are the only instances of sub-processes which follow a read on the tail pointer.

If the g_i-thread has the form $\mathsf{EnqRdTN}_{\mathsf{cas}}\langle\phi_e\rangle$, it reduces to $\mathsf{EnqRdTP}_{\mathsf{cas}}\langle\phi'_e\rangle$. Since the tail pointer has been swung forward, the g_i-thread has an outdated copy which may not refer to the last node in the linked list: that is, the successor of $\phi'_e(last)$ may not be $null$. Hence, $\mathsf{EnqRdTP}_{\mathsf{cas}}\langle\phi'_e\rangle$ is reduced to $\mathsf{EnqSwRec}_{\mathsf{cas}}\langle\phi''_e\rangle$, where we try to swing the tail forward (unsuccessfully because its value has already been updated), then we go back to the beginning of the loop and read the updated value of the tail pointer, thus reaching Q'''.

If the g_i-thread of Q'' has the form $\mathsf{DeqRdHN}_{\mathsf{cas}}\langle\phi_d\rangle$, it reduces to $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$, where there are two cases: either $\phi'_d(hn) = \phi'_d(tn)$ or not (that is, either head and tail

pointed to the same node or not). In the former case, we need to further consider if the tail points to the last node in the queue. But since it is out of date, it may not point to the last node. Then $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$ reduces to $\mathsf{DeqSw}_{\mathsf{cas}}\langle\phi''_d\rangle$, goes back to the beginning of the loop, and eventually we reach Q'''. Now, in the latter case, $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$ is reduced to $\mathsf{DeqRdNext}_{\mathsf{cas}}\langle\phi'''_d\rangle$. Since the following reductions are independent of the value of the tail pointer, we have reached a process Q''' which satisfies the invariant: $Q' \overset{\mathsf{nc}(g_i)}{\longmapsto}{}^{*} Q''' \overset{\mathsf{nc}(g_i)}{\longmapsto}{}^{*} R'$ or $Q' \overset{\mathsf{nc}(g_i)}{\longmapsto}{}^{*} R' \overset{\mathsf{nc}(g_i)}{\longmapsto}{}^{*} Q'''$.

Let x be the head pointer. Then $\ell(g_j)$ is actually a commit molecular action:

$$Q \overset{\mathsf{nc}(g_i)}{\longmapsto} \overset{\mathsf{com}(g_j)}{\longmapsto} \overset{\mathsf{norm}(g_j)}{\longmapsto} Q''$$

Note that in Q'', the g_i-thread is of the form $\mathsf{DeqRdT}_{\mathsf{cas}}\langle\phi_d\rangle$ (for a proper substitution ϕ_d), as this is the only LMF which follows a read on the head pointer. In two (molecular) steps, we reach $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$. By contradiction assume that $\phi'_d(hn) = \phi'_d(tn)$. Currently, $\phi'_d(hn)$ is outdated since the g_j-thread has swung the head forward, while $\phi'_d(tn)$ is up-to-date since the tail has been read after the completion of the g_j-thread. This implies that the head points to a later node than the tail, which contradicts Prop. 4.6. Then $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$ is reduced to $\mathsf{DeqRdNext}_{\mathsf{cas}}\langle\phi''_d\rangle$, which is reduced to $\mathsf{DeqCom}_{\mathsf{cas}}\langle\phi'''_d\rangle$. The next cas operation brings us back to the beginning of the loop, since the value of the head pointer is outdated. Then eventually we reach Q'''.

Let x be the last pointer in the linked list. Again, $\ell(g_j)$ is a commit:

$$Q \overset{\mathsf{nc}(g_i)}{\longmapsto} \overset{\mathsf{com}(g_j)}{\longmapsto} \overset{\mathsf{norm}(g_j)}{\longmapsto} Q''$$

Note that in Q, the g_i-thread is either of the form $\mathsf{EnqRdTP}_{\mathsf{cas}}\langle\phi_e\rangle$ or of the form $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi_d\rangle$ (for proper substitutions ϕ_e and ϕ_d), because these are the only LMFs at which we perform a read on some successor pointer. In the first case, x must be $null$ because the following cas is a commit. Then in Q'', the g_i-thread is of the form $\mathsf{EnqCom}_{\mathsf{cas}}\langle\phi_e\rangle$. However the next cas operation fails to modify the pointer because it has already been modified. Then we go back to the beginning of the loop and reach Q'''.

In the case of $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi_d\rangle$, we are assuming that $\phi_d(h) = \phi_d(t)$, given that the g_j-thread may perform a commit cas action which adds a successor to $\phi_d(hn)$. However, as the read action in the g_i-thread moves left of the commit cas action, it becomes itself a commit. But we have already excluded this case. Then we are done. □

The latter case of Proposition 4.10 (nc-up) requires a simpler statement.

Lemma G.16. *Let P be a queue process such that $P \overset{\mathsf{norm}(g_j)}{\longmapsto} P' \overset{\mathsf{sw}(g_i)}{\longmapsto} R$. Then $P \overset{\mathsf{sw}(g_i)}{\longmapsto} \overset{\mathsf{norm}(g_j)}{\longmapsto} R$.*

Proof. We need to consider each possible local permutation and see if there are critical cases. First of all, as long as a tail-swinging operation is permuted with transitions where the tail is not involved, there is no problem. When it follows another occurrence of tail-swinging operation, we just apply Lemma G.14.

Then let the tail-swinging in the g_i-thread follow a $read$ on the tail in the g_j-thread:

$$Q \overset{\mathsf{nc}(g_j)}{\longmapsto} \overset{\mathsf{sw}(g_i)}{\longmapsto} Q' \overset{\mathsf{norm}(g_j)}{\longmapsto} Q''$$

If the cas does not actually modify the tail, the permutation has no effect on the $read$. Then we assume that the tail *is* modified. After the permutation, we obtain:

$$Q \xmapsto{\mathsf{sw}(g_i)} \xmapsto{\mathsf{nc}(g_j)} Q'''$$

where the normalisation sequence is not indicated because in Q''' the g_j-thread has read a value of the tail pointer which is up to date, whereas in Q' the read value was outdated. Through manipulation of $Q' \xmapsto{\mathsf{norm}(g_j)} Q''$, we shall show that $Q''' \xmapsto{\mathsf{norm}(g_j)} Q''$. Consider the g_j-thread in Q'. It may be either of the form $\mathsf{EnqRdTN}_{\mathsf{cas}}\langle\phi_e\rangle$ or of the form $\mathsf{DeqRdHN}_{\mathsf{cas}}\langle\phi_d\rangle$ (for proper substitutions ϕ_e and ϕ_d), as these are the only instances of sub-processes which follow a read on the tail pointer.

Let it have the form $\mathsf{EnqRdTN}_{\mathsf{cas}}\langle\phi_e\rangle$. Then it reduces to $\mathsf{EnqRdTP}_{\mathsf{cas}}\langle\phi'_e\rangle$. Since $\phi'_e(t)$ is outdated, $\phi'_e(tn)$ may not be the last node. Hence we reach $\mathsf{EnqSwRec}_{\mathsf{cas}}\langle\phi''_e\rangle$, where we try to swing the tail forward (unsuccessfully because the g_i-thread already did it). Then we go back to the beginning of the loop and read the updated value of the tail. At this point all the substitutions are as in Q'''. Therefore $Q''' \xmapsto{\mathsf{norm}(g_j)} Q''$.

If the g_j-thread of Q' has the form $\mathsf{DeqRdHN}_{\mathsf{cas}}\langle\phi_d\rangle$, it reduces to $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$. First let $\phi'_d(hn) = \phi'_d(tn)$. But just as above, $\phi'_d(tn)$ may not be the last node. Then we reach $\mathsf{DeqSw}_{\mathsf{cas}}\langle\phi''_d\rangle$ and go back to the beginning of the loop, eventually reaching Q''' again (note in particular that the value of the head in Q''' is up to date as well). If $\phi'_d(hn) \neq \phi'_d(tn)$, $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi'_d\rangle$ reduces to $\mathsf{DeqRdNext}_{\mathsf{cas}}\langle\phi''_d\rangle$. Since the following reductions are independent of the value of the tail pointer, we eventually reach a process Q'''', which would have been reached from Q''' anyway. Therefore $Q''' \xmapsto{\mathsf{norm}(g_j)} Q''$. □

The above lemmas lead to the proof of Proposition 4.10 (nc-up).

Proof of Proposition 4.10(1): (nc-up)
Let $P \xmapsto{\mathsf{norm}(g_j)} P' \xmapsto{\mathsf{nc}(g_i)} R$. We show the claim by cases of $P' \xmapsto{\mathsf{nc}(g_i)} R$.

Let $P' \xmapsto{\mathsf{nc}(g_i)} R$ be the synchronisation of either $\mathsf{EnqReq}_{\mathsf{cas}}\langle r, v, g\rangle$ or $\mathsf{DeqReq}_{\mathsf{cas}}\langle r, g\rangle$ with the initial branching in $\mathsf{CQ}(r, h, t)$. Note that such synchronisations do not prevent the occurrence of other transitions nor they modify the linked list. Then $P \xmapsto{\mathsf{nc}(g_j)} P'' \xmapsto{\mathsf{norm}(g_j)} R$, where $P'' \xmapsto{\mathsf{norm}(g_j)} R$ consists of exactly the same transitions as $P \xmapsto{\mathsf{norm}(g_j)} P'$.

The case of $P' \xmapsto{\mathsf{nc}(g_i)} R$ being a $read$ operation has been shown in Lemma G.15 and Lemma G.16 gives the case of a tail-swinging operation. Then we are done. □

The last critical permutation consists in moving a cas on the tail right of a commit action. In the case of an enqueue commit, the proof relies on the observation that if such transitions occur one after the other, then the cas is ineffective. In the case of a dequeue commit, we show that the two transitions do not interfere with each other.

Proof of Proposition 4.10(2): (sw-com)

Let $P \xmapsto{\mathsf{sw}(g_i)} P' \xmapsto{\mathsf{com}(g_j)} R$. Note that i and j must be different, because there is no occurrence of commit action which immediately follows a cas on the tail in the same thread. We show the claim by cases of $P' \xmapsto{\mathsf{com}(g_j)} R$.

Let $P' \xmapsto{\mathsf{com}(g_j)} R$ be an enqueue commit action. Then, at P', the g_j-thread has the form $\mathsf{EnqCom}_{\mathsf{cas}}\langle\phi_e\rangle$ (for a proper substitution ϕ_e). In order for $P' \xmapsto{\mathsf{com}(g_j)} R$ to be a commit, $\phi_e(t)$ must necessarily refer to the last node in the linked list. Then $P \xmapsto{\mathsf{sw}(g_i)} P'$ must have been ineffective because the tail was pointing to the last node in P as well. By permuting the two transitions:

$$P \xmapsto{\mathsf{com}(g_j)} R' \xmapsto{\mathsf{sw}(g_i)} R''$$

we have the intermediate process R', where the tail is not pointing to the last node anymore. Note however that the value of the tail in the g_i-thread is outdated, because it was read before. Then $R' \xmapsto{\mathsf{sw}(g_i)} R''$ is still ineffective. Then R'' and R are the same process.

There are two cases of dequeue commit action. First let $P' \xmapsto{\mathsf{com}(g_j)} R$ correspond to the reduction of $\mathsf{DeqRdHP}_{\mathsf{cas}}\langle\phi_d\rangle$ (for a proper substitution ϕ_d), reading $null$ as value of the last pointer in the linked list. Note that $\phi_d(t) = \phi_d(h)$ refers to the last node, since P' is in general form (Proposition 4.6(a)). Then, just as above, the tail swinging operation $P \xmapsto{\mathsf{sw}(g_i)} P'$ must have been ineffective. Then by permuting we obtain the same final process:

$$P \xmapsto{\mathsf{com}(g_j)} R' \xmapsto{\mathsf{sw}(g_i)} R$$

The other case of dequeue commit action corresponds to a cas on the head pointer, where the g_j-thread at P' must have the form $\mathsf{DeqCom}_{\mathsf{cas}}\langle\phi_d\rangle$ (for a proper substitution ϕ_d). Then necessarily we must have $\phi_d(h) \neq \phi_d(t)$. Since $P' \xmapsto{\mathsf{com}(g_j)} R$ is a commit, $\phi_d(h)$ is up-to-date. Then, both in P and in P', the head refers to an earlier node than the tail (Proposition 4.6(a)). Then the commit action and the tail swinging operation do not interfere with each other and we have:

$$P \xmapsto{\mathsf{com}(g_j)} R' \xmapsto{\mathsf{sw}(g_i)} R$$

Then we are done. □

G.4 Weak Bisimulation $\mathcal{R}_{\mathsf{cas}}$

The following result says that if a queue process is related to an abstract queue, then they share a common history in some (weak) sense. It is required in the proof of Lemma 4.13 (among others)

Lemma G.17. *Let $Q\,\mathcal{R}_{\mathsf{cas}}q$, then $\mathsf{CQemp}(r)$ and $\mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle)$ admit the following transition sequences:*

$$\mathsf{CQemp}(r) \xmapsto{\ell_1} \ldots \xmapsto{\ell_n} Q \qquad \Longleftrightarrow \qquad \mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle) \xrightarrow{\widehat{\ell_1}} \ldots \xrightarrow{\widehat{\ell_n}} q$$

Proof. The base case is $Q = \mathsf{CQemp}(r)$ and $q = \mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle)$. In this case, both $\mathsf{CQemp}(r)$ and $\mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle)$ can reach themselves through the empty sequence.

Otherwise, there have to be P and p such that $P\,\mathcal{R}_{\mathsf{cas}}p$, $P \xmapsto{\ell_P} Q$ and $p \xrightarrow{\widehat{\ell_P}} q$, by definition of R. Then by iterating this reasoning, we have:

$$\mathsf{CQemp}(r) \xmapsto{\ell_1} \ldots \xmapsto{\ell_n} Q \qquad \Longleftrightarrow \qquad \mathsf{AQ}(r, \langle\emptyset, \varepsilon, \emptyset\rangle) \xrightarrow{\widehat{\ell_1}} \ldots \xrightarrow{\widehat{\ell_n}} q$$

which concludes the proof. □

G.5 Construction and Bisimilarity of $\mathcal{R}_{\mathsf{lock}}$

In this subsection we define the relation $\mathcal{R}_{\mathsf{lock}}$ between lock-based queue processes and abstract queues, and we show that it is a bisimulation. The proof has the same structure as the analogous proof we gave for $\mathcal{R}_{\mathsf{cas}}$, although it is much simpler in many parts.

As we have done for cas-queue processes, we use lin-nfs of lock-based queue processes, noting that the initial process is in linear formal form.

Convention G.18 (lock-queue process in lin-nf). *Henceforth, a "(lock-)queue process" always denotes a derivative of a typed transition sequence from* $\mathsf{LQemp}(r)$ *which is also in lin-nf.*

Proposition G.19. *Any lock-queue process under Convention G.18 can be reached from the empty queue by a sequence of molecular actions.*

Proof. By Proposition G.2 and because, by construction, $\mathsf{LQemp}(r)$ is in lin-nf. □

A lock-queue process has always a specific syntactic form. As we did for operations of cas-queue processes, we first define the "local molecular forms" for operations of lock-queue processes. Below, we write $\mathsf{Enq}_{\mathrm{lck}}^{(n)}\langle v_1,\ldots,v_m\rangle$ to indicate the derivative of the enqueue process $\mathsf{Enqueue}_{\mathrm{cas}}\langle t,v,g\rangle$ which has reached Line n, with v_1, …,v_m values instantiated in this order (omitting the substitutions that are not used anymore), similarly for $\mathsf{Deq}_{\mathrm{lck}}^{(n)}\langle v_1,\ldots,v_m\rangle$.

Definition G.20 (local molecular form, LMF). A *local molecular form, or LMF, for an enqueue/dequeue operation* is a process in one of the forms in Figure 4. In lines (2,3,4), we fix $C[\,\cdot\,] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, nd)([\,\cdot\,] \mid \mathsf{LENode}(nd,v))$. Also, g in each line is called the *continuation name* of the process.

We say that a LMF for an enqueue/dequeue operation is:

- a *free request*, if it is of either form $\mathsf{EnqReq}_{\mathrm{lck}}\langle r,v,g\rangle$ or $\mathsf{DeqReq}_{\mathrm{lck}}\langle r,g\rangle$;
- *pending*, if it is of either form $\mathsf{EnqCom}_{\mathrm{lck}}\langle t,v,g\rangle$ or $\mathsf{DeqCom}_{\mathrm{lck}}\langle h,t,g\rangle$;
- an *answer* if it is of either form $\mathsf{EnqAns}_{\mathrm{lck}}\langle g\rangle$, $\mathsf{DeqAnsNull}_{\mathrm{lck}}\langle g\rangle$ or $\mathsf{DeqAns}_{\mathrm{lck}}\langle v,g\rangle$;
- *critical* if it is of either of the following forms:
  - $\mathsf{EnqRdT}_{\mathrm{lck}}\langle y',t,v,g\rangle$,
  - $\mathsf{EnqWrNext}_{\mathrm{lck}}\langle tn,y',t,v,g\rangle$,
  - $\mathsf{EnqSwT}_{\mathrm{lck}}\langle y',t,nd,g\rangle$,
  - $\mathsf{DeqRdH}_{\mathrm{lck}}\langle y',h,t,g\rangle$,
  - $\mathsf{DeqRdT}_{\mathrm{lck}}\langle hn,y',h,t,g\rangle$,
  - $\mathsf{DeqRdHn}_{\mathrm{lck}}\langle hn,y',h,g\rangle$ or
  - $\mathsf{DeqSwH}_{\mathrm{lck}}\langle v,nxtHd,y',h,g\rangle$.

Intuitively, a critical LMF indicates that the operation is in the critical section.

Using local molecular forms, we define general forms of lock-queue processes in the expected way:

$$
\begin{aligned}
\mathsf{EnqReq}_{\mathrm{lck}}\langle r, v, g\rangle &\stackrel{\text{def}}{=} \overline{r} \oplus \mathtt{enq}\langle v, g\rangle \\
\mathsf{EnqCom}_{\mathrm{lck}}\langle t, v, g\rangle &\stackrel{\text{def}}{=} \bar{l}(c)c(y).C[\mathsf{Enq}^{(1)}_{\mathrm{lck}}\langle t, nd, g\rangle] \\
\mathsf{EnqRdT}_{\mathrm{lck}}\langle y', t, v, g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(2)}_{\mathrm{lck}}\langle y', t, nd, g\rangle] \\
\mathsf{EnqWrNext}_{\mathrm{lck}}\langle tn, y', t, v, g\rangle &\stackrel{\text{def}}{=} C[\mathsf{Enq}^{(3)}_{\mathrm{lck}}\langle tn, y', t, nd, g\rangle] \\
\mathsf{EnqSwT}_{\mathrm{lck}}\langle y', t, nd, g\rangle &\stackrel{\text{def}}{=} \mathsf{Enq}^{(4)}_{\mathrm{lck}}\langle y', t, nd, g\rangle \\
\mathsf{EnqAns}_{\mathrm{lck}}\langle g\rangle &\stackrel{\text{def}}{=} \mathsf{Enq}^{(5)}_{\mathrm{lck}}\langle g\rangle \stackrel{\text{def}}{=} \overline{g} \\
\mathsf{DeqReq}_{\mathrm{lck}}\langle r, g\rangle &\stackrel{\text{def}}{=} \overline{r} \oplus \mathtt{deq}\langle g\rangle \\
\mathsf{DeqCom}_{\mathrm{lck}}\langle h, t, g\rangle &\stackrel{\text{def}}{=} \bar{l}(c)c(y).\mathsf{Deq}^{(1)}_{\mathrm{lck}}\langle h, t, g\rangle \\
\mathsf{DeqRdH}_{\mathrm{lck}}\langle y', h, t, g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(1)}_{\mathrm{lck}}\langle y', h, t, g\rangle \\
\mathsf{DeqRdT}_{\mathrm{lck}}\langle hn, y', h, t, g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(2)}_{\mathrm{lck}}\langle hn, y', h, t, g\rangle \\
\mathsf{DeqAnsNull}_{\mathrm{lck}}\langle g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(4)}_{\mathrm{lck}}\langle g\rangle \stackrel{\text{def}}{=} \overline{g}\langle null\rangle \\
\mathsf{DeqRdHn}_{\mathrm{lck}}\langle hn, y', h, g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(5)}_{\mathrm{lck}}\langle hn, y', h, g\rangle \\
\mathsf{DeqSwH}_{\mathrm{lck}}\langle v, nxtHd, y', h, g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(6)}_{\mathrm{lck}}\langle v, nxtHd, y', h, g\rangle \\
\mathsf{DeqAns}_{\mathrm{lck}}\langle v, g\rangle &\stackrel{\text{def}}{=} \mathsf{Deq}^{(7)}_{\mathrm{lck}}\langle v, g\rangle \stackrel{\text{def}}{=} \overline{g}\langle v\rangle
\end{aligned}
$$

Fig. 4. Local molecular forms for lock-queue processes

Definition G.21 (general form). A queue process P is in *general molecular form* or in *general form* for short, when either:

$$P \equiv (\nu h, t, nd_0, \ldots, nd_n, l)(\ \mathsf{Mtx}\langle l\rangle \mid \mathsf{LQ}(r, h, t, l) \mid \textstyle\prod_{1\le i\le m} P_i \mid LL\ )$$

in which case we say that P is *unlocked*, or:

$$P \equiv (\nu h, t, nd_0, \ldots, nd_n, l, l')(\ l'.\mathsf{Mtx}\langle l\rangle \mid \mathsf{LQ}(r, h, t, l) \mid \textstyle\prod_{1\le i\le m} P_i \mid LL\ )$$

in which case we say that P is *locked*. In any case P is typed under $r : \&\{\mathtt{enq}(\alpha \uparrow ()), \mathtt{deq}(\uparrow(\alpha)\},\ \{g_i :\uparrow (\vec{\alpha})\}_{1\le i\le n}$, for some α. Moreover, each P_i is in LMF and contains a single free occurrence of a name g_i (we say P_i is a *g_i-thread*, or simply *thread*, *of* P). Finally, LL ("*linked list sub-process*") has the following form:

$$\mathsf{LPtr}(h, nd_H) \mid \mathsf{LPtr}(t, nd_T) \mid \textstyle\prod_{1\le i\le n} \mathsf{LNode}(nd_i, v_i, nd_{i+1})$$

where we have $nd_{n+1} = null$, $0 \le H \le T = n$.

Again, the two most basic properties of general forms apply also to lock-queues.

Proposition G.22. *Let $\Gamma \vdash P$ be a lock-queue process in general form.*

1. *Exactly one output in each thread of P is ready in P and these outputs cover all the outputs that are ready in P.*
2. *If $\Gamma \vdash P \stackrel{\ell}{\to} \Gamma' \vdash P'$ where ℓ is an input or an output, then P' is also in general form.*

Proof. Same as the proof of Proposition G.11. □

By using these properties, we establish the invariants of lock-queue processes.

Proposition G.23 (invariants of lock-queue processes). *Let P be a lock-queue process (hence by Convention G.18 a lin-nf). Then:*

(a) *P is in general form.*
(b) *If P is locked then one and only one thread P_i of P is critical.*
(c) *If P is unlocked then no thread of P is critical.*

Proof. Suppose $\mathsf{LQemp}(r) \stackrel{\ell_1}{\longmapsto} .. \stackrel{\ell_n}{\longmapsto} P$. We establish all three points by a single induction on n.

(Base Step.) This is when $n = 0$ and $P = \mathsf{LQemp}(r)$.

(a) Immediate since the linked list has only the dummy node and there are no threads.
(b,c) Vacuous since no thread exists.

(Inductive Step.) We assume the result holds for $n = m$, and show the same holds for $n = m + 1$. So suppose:

$$\mathsf{LQemp}(r) \stackrel{\ell_1}{\longmapsto} .. \stackrel{\ell_m}{\longmapsto} P \stackrel{\ell_{m+1}}{\longmapsto} P'.$$

By the typing, ℓ_{m+1} can be either the input of a request, the output of an answer, or a τ-action. The first two cases are by Proposition G.22 (2). If $\ell_{m+1} = \boldsymbol{\tau}$, then by Proposition G.22 (1), the initial τ-action implies that the prefix of one of the threads, say P_i, reduces. We reason on the shape of P_i.

Case $P_i = \mathsf{EnqReq}_{\mathbf{lck}}\langle r, v, g\rangle$ **(free request):** Then the involved reduction is:

$$\mathsf{EnqReq}_{\mathrm{lck}}\langle r, v, g\rangle|\mathsf{LQ}(r, h, t, l) \longrightarrow \mathsf{EnqCom}_{\mathrm{lck}}\langle t, v, g\rangle|\mathsf{LQ}(r, h, t, l)$$

reducing to a LMF without changing the contents of h, t and l, nor the linked-list. Hence **(a)**, **(b)** and **(c)** immediately hold.

Note that the same reasoning applies as well to $\mathsf{DeqReq}_{\mathrm{lck}}\langle r, g\rangle$, the other case of free request.

Case $P_i = \mathsf{EnqCom}_{\mathbf{lck}}\langle t, v, g\rangle$ **(pending):** the involved reduction is:

$$\mathsf{EnqCom}_{\mathrm{lck}}\langle t, v, g\rangle|l(x).\overline{x}(y)y.\mathsf{Mtx}\langle l\rangle \stackrel{\tau}{\longmapsto} (\nu l')(\mathsf{EnqRdT}_{\mathrm{lck}}\langle l', t, v, g\rangle|l'.\mathsf{Mtx}\langle l\rangle)$$

reducing an unlocked general form to a locked one (then **(a)** and **(c)** still hold). Moreover, by induction hypothesis, no thread of P was critical since P is unlocked. Then **(b)** also holds, as exactly one critical thread was added. Moreover, the same reasoning applies as well to $\mathsf{DeqCom}_{\mathrm{lck}}\langle h, t, g\rangle$, the other case of pending process.

Case $P_i = \mathsf{DeqRdT}_{\mathbf{lck}}\langle hn, y', h, t, g\rangle$ **(critical):** The molecular transition ℓ_{m+1} first reads the name tn contained in t, then compares it with hn. If $hn = tn$ the involved reduction is:

$$(\nu y')(\mathsf{DeqRdT}_{\mathrm{lck}}\langle hn, y', h, t, g\rangle|y'.\mathsf{Mtx}\langle l\rangle) \stackrel{\tau}{\longmapsto} \mathsf{DeqAnsNull}_{\mathrm{lck}}\langle g\rangle|\mathsf{Mtx}\langle l\rangle$$

where P goes from locked to unlocked and, accordingly, the thread is no more critical. Hence, each of **(a)**, **(b)** and **(c)** is satisfied. The same reasoning applies also to the cases of $\mathsf{EnqSwT}_{\mathrm{lck}}\langle y', t, nd, g\rangle$ and $\mathsf{DeqSwH}_{\mathrm{lck}}\langle v, nxtHd, y', h, g\rangle$, where a critical thread is transformed into a non-critical one.

On the other hand, if $hn \neq tn$ the involved reduction is:

$$\mathsf{DeqRdT}_{\mathrm{lck}}\langle hn, y', h, t, g\rangle \overset{\tau}{\longmapsto} \mathsf{DeqRdHn}_{\mathrm{lck}}\langle hn, y', h, g\rangle$$

where P stays locked and, accordingly, the thread stays critical. Hence, each of **(a)**, **(b)** and **(c)** is satisfied again. Note that the same reasoning applies to each transition after which a critical thread stays critical, namely the cases of $\mathsf{EnqRdT}_{\mathrm{lck}}\langle y', t, v, g\rangle$, $\mathsf{EnqWrNext}_{\mathrm{lck}}\langle tn, y', t, v, g\rangle$, $\mathsf{DeqRdH}_{\mathrm{lck}}\langle y', h, t, g\rangle$ and $\mathsf{DeqRdHn}_{\mathrm{lck}}\langle hn, y', h, g\rangle$.

The only shape of P_i which has not been analysed is that of the answer. However, answers may not be reduced by τ-transitions, as they may only produce outputs. Then we are done. □

As in the analogous definition for cas-queue processes, we say that a LMF P_i in a lock-based queue process is a *local normal form (LNF)* when it is either pending or an answer. The notions of non-commit action ($P \overset{\mathsf{nc}}{\longmapsto} Q$ and $P \overset{\mathsf{nc}(g)}{\longmapsto} Q$), and of local normalisation ($P \overset{\mathsf{norm}(g_1,\ldots,g_n)}{\longmapsto} Q$) for lock-queue processes, are defined exactly as those for cas-queue processes. We also distinguish *committing* local normalisations (those which contain a commit action) from *non-committing* ones. The definition of normal form is even simplified.

Definition G.24 (normal form). A queue process P in general form is in *normal form* when all of its threads are in LNF.

It follows directly from the definition that when P is in normal form, it is unlocked and none of its threads is critical.

Now we show that a normal form can always be reached.

Lemma G.25. *For a lock-queue process P, $P \overset{\mathsf{nc}}{\longmapsto}{}^{*} P'$ such that P' is in normal form.*

Proof. By Proposition G.23, P is in general form and at most one of its threads is critical. Then each other thread is either a free request, a pending process, or an answer. In the former case, a single reduction will make it a pending process, while in the latter two cases, it is already in LNF by definition. Finally, a critical process is reduced to an answer simply by following the reduction sequence line by line. □

The definitions of the relation $\mathcal{R}_{\mathsf{lock}}$ (between lock-queue processes and abstract queues) is also identical to that of $\mathcal{R}_{\mathsf{cas}}$ (between cas-queue processes and abstract queues). As for the definition of $\widehat{\mathcal{R}_{\mathsf{lock}}}$, let P be a lock-queue process and p an abstract queue:

$$P \,\widehat{\mathcal{R}_{\mathsf{lock}}}\, p \quad \overset{\text{def}}{\Leftrightarrow} \quad P \text{ normal form and } p = \mathsf{AQ}(r, \langle \mathbf{req}(P), \mathbf{val}(P), \mathbf{ans}(P)\rangle)$$

Although this looks just like the definition of $\widehat{\mathcal{R}_{\mathsf{cas}}}$, there are minor differences in the way the functions $\mathbf{req}(-)$, $\mathbf{val}(-)$ and $\mathbf{ans}(-)$ are defined. Nonetheless, these differences simply reflect the different shapes of lock-queue and cas-queue processes, as expected. Then we omit their definitions.

The modular framework allows us to re-use even proofs given for cas-queue processes in order to show analogous statements for lock-queue processes. As in the following lemma, where both the statement and the proof are identical (up to exchange of cas-queue process with lock-queue process) to those of Lemma G.17.

Lemma G.26. *Let $Q \mathcal{R}_{\mathsf{lock}} q$, then $\mathsf{LQemp}(r)$ and $\mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset \rangle)$ admit the following transition sequences:*

$$\mathsf{LQemp}(r) \xmapsto{\ell_1} \ldots \xmapsto{\ell_n} Q \qquad \Longleftrightarrow \qquad \mathsf{AQ}(r, \langle \emptyset, \varepsilon, \emptyset \rangle) \xrightarrow{\widehat{\ell_1}} \ldots \xrightarrow{\widehat{\ell_n}} q$$

The following result is analogous to Lemma 4.13.

Lemma G.27. *Let P be a lock-based queue process in normal form. Then:*

1. *$P \mathcal{R}_{\mathsf{lock}} p$ implies $P \widehat{\mathcal{R}_{\mathsf{lock}}} p$.*
2. *If $P \widehat{\mathcal{R}_{\mathsf{lock}}} p$ then $P \xmapsto{\ell}$ iff $p \xrightarrow{\ell}$, with ℓ being a commit or visible.*

Proof. For the proof of (1), the key part is to show linearisability. Note that the molecular actions that form the critical section of the g_i-thread can only be interleaved with the following kinds of transitions from some g_j-thread (with $i \neq j$):

- inputs, which can be moved all the way left by Proposition 2.3(3);
- outputs, which can be moved all the way right by Proposition 2.3(3);
- internal reductions of either of the following two forms:

$$\mathsf{EnqReq}_{\mathsf{lck}}\langle r, v, g\rangle | \mathsf{LQ}(r, h, t, l) \longrightarrow \mathsf{EnqCom}_{\mathsf{lck}}\langle t, v, g\rangle | \mathsf{LQ}(r, h, t, l)$$
$$\mathsf{DeqReq}_{\mathsf{lck}}\langle r, g\rangle | \mathsf{LQ}(r, h, t, l) \longrightarrow \mathsf{DeqCom}_{\mathsf{lck}}\langle h, t, g\rangle | \mathsf{LQ}(r, h, t, l)$$

  Since both of these only affect the g_j-thread, they are independent of any molecular action on the g_i-thread. Hence, they can be moved right until just before the g_j-thread's critical section.

By applying the moves described above everywhere, we obtain the linearised sequence. The rest is as in the proof of Lemma 4.13(1).

The proof of (2) is also essentially the same as the proof of Lemma 4.13(2).

The proof that $\mathcal{R}_{\mathsf{lock}}$ is a bisimulation is also identical to the analogous result shown for $\mathcal{R}_{\mathsf{cas}}$ (Proposition 4.14). Then we state:

Proposition G.28. *$\mathcal{R}_{\mathsf{lock}}$ is a weak bisimulation.*

G.6 Theorem 4.16 (Correctness)

Proof of Theorem 4.16 We just compose $\mathcal{R}_{\mathsf{lock}}$ and the inverse of $\mathcal{R}_{\mathsf{cas}}$, to obtain a relation associating cas-queue processes to lock-based ones. Projecting on two concrete queues, we obtain a bisimulation which we write $\mathcal{R}_{\mathsf{lock,cas}}$. □

G.7 cas Queue Processes preserve the General Form in the presence of Failures

In the bisimilarity proofs, failures were not considered because the goal was to show algorithmic correctness: assuming that no errors are introduced by the implementation, it is ensured that, at *any* point, a computation *can* be successful. However, progress properties must be ensured independently of some stalling computation. Then the operational semantics is augmented with failing reductions.

The following statement extends Prop. 4.6(a), by admitting failures. The only reason why the statement has to be weakened (considering general forms up to $\equiv$) is that now we need to do garbage collection for the irrelevant sub-processes left over by previous failures.

Lemma G.29. *Let P' be a* cas *queue process. Then $P' \equiv P$ and P is in general form.*

Proof. We just need to take the proof of Proposition 4.6 (a) and add the case of a failing reduction to the inductive step.

Let R' be a cas queue process such that $R' \equiv R$ and R is in general form:

$$R' \equiv R = (\nu\ h, t, \vec{nd})(\mathsf{CQ}(r, h, t) \mid LL_R \mid \prod_{1 \leq i \leq m} P_i)$$

$R' \equiv R$ implies that, for any transition $R' \stackrel{\ell}{\mapsto} P'$, there is a transition $R \stackrel{\ell}{\mapsto} P''$, such that $P' \equiv P''$. Then we shall consider $R \stackrel{\ell}{\mapsto} P''$, rather than the original transition from R'.

So assume that $R \stackrel{\ell}{\mapsto} P''$ is a failing reduction. Since a failing reduction may not reduce linear prefixes, it may not have taken place in the linked-list sub-process LL_R. Then it must have taken place inside some thread P_j (for $1 \leq j \leq m$). By definition, P_j is in local molecular form. Note that for any case of LMF, a failing reduction is such that:

$$P_j \longrightarrow P'_j \equiv \mathbf{0}$$

then:

$$P'' \equiv P = (\nu\ h, t, \vec{nd})(\mathsf{CQ}(r, h, t) \mid LL_R \mid \prod_{1 \leq i \leq j-1} P_i \mid \prod_{j+1 \leq i \leq m} P_i)$$

and P is in general form. By transitivity of $\equiv$, $P' \equiv P$. □

G.8 Global Progress in Lock-queues

As we have mentioned in § 4, the lock-based queue $\mathsf{LQemp}(r)$ is not non-blocking and the reason is that it is not resilient. However, by allowing non-failing transition sequences only, we can show that $\mathsf{LQemp}(r)$ is weakly wait free, as we do below. We maintain the assumption (motivated in § G.5) that lock-queue processes are in lin-nf, hence by Proposition G.23 (a), they are also in general form.

Proposition G.30 (WWF). $\mathsf{LQemp}(r)$ *is WWF.*

Proof. Consider the fair non-failing transition sequence:

$$\Phi : \Gamma_Q \vdash \mathsf{LQemp}(r) \longrightarrow \Gamma_P \vdash P \longrightarrow$$

and let $g \in \texttt{allowed}(\Gamma_P)$. Then an input has occurred in the first part of the sequence generating an enqueue/dequeue request, of either form $\mathsf{EnqReq}_{\text{lck}}\langle r, v, g\rangle$ or $\mathsf{DeqReq}_{\text{lck}}\langle r, g\rangle$. Then P must still contain a thread P^g in LMF where g appears, since P is in general form. Note in particular, that g disappears only when an output on g occurs, but when that happens g is no more allowed. Then an output on g may not have happened before P. We start by assuming P^g is in one of the above two forms and by following the sequence of reductions, we shall provide a reasoning that encompasses the other cases as well.

Without loss of generality, consider $P^g = \mathsf{EnqReq}_{\text{lck}}\langle r, v, g\rangle$, the same reasoning applies also to the dequeue LMFs. P^g is waiting for a synchronisation on r with $\mathsf{LQemp}(r)$, which is enabled because r is active in $\mathsf{LQemp}(r)$. Then by fairness it shall occur, consuming P^g and adding the pending process $R^g = \mathsf{EnqCom}_{\text{lck}}\langle t, v, g\rangle$. Now R^g is waiting for a synchronisation with the initial branching on l in the mutex agent sub-process. However, l may not be active there if some other thread P_j has already taken the lock and not released it yet. Nonetheless, we can be sure that P_j will eventually release the lock in Φ, since Φ is fair and non-failing. Then the synchronisation between R^g and the mutex agent shall be enabled and by fairness it shall occur. All the following reductions are synchronisations with node or pointer agents, whose initial subject is always active. Then this sequence of reductions eventually leads to the answer $\mathsf{EnqAns}_{\text{lck}}\langle g\rangle$, where an output on g is enabled. Then by fairness it shall occur. □

H Proofs for Section 5 (Separation Results)

The separation result consists of a few intermediate results, as we have seen in § 4. The proof of Lemma 5.7 relies on the fact that for any transition sequence there is a sequence of molecular actions with the same weak trace and set of blocked outputs. Formally:

Lemma H.1. *Let $\Phi : P \longrightarrow$ be a fair and finitely failing sequence of transitions. Then there is a fair and finitely failing sequence Ψ of molecular actions from P such that $\langle\widehat{\Phi}, \texttt{blocked}(\Phi)\rangle = \langle\widehat{\Psi}, \texttt{blocked}(\Psi)\rangle$.*

Proof. We do the proof by induction on the length of Φ. Throughout the induction, we also satisfy the invariant that any transition enabled after Φ either occurs in Ψ or is enabled after Ψ.

Let Φ be empty. Note that no conditional, output or linear input may be enabled after Φ, since Φ is fair. Then Ψ is Φ and it satisfies both the claim and the invariant.

Now assume that both the claim and the invariant hold for a transition sequence $\Phi' : P \longrightarrow Q$ and a sequence of molecular actions $\Psi' : P \overset{\ell_1}{\longmapsto} \ldots \overset{\ell_n}{\longmapsto} R$, and let Φ consist of Φ' followed by $Q \xrightarrow{\ell}$.

If $Q \xrightarrow{\ell}$ is an internal transition reducing a redex that has already been reduced in Ψ', we set $\Psi = \Psi'$. Since ℓ is neither visible nor a failing reduction, the claim is still satisfied. Moreover, any transition which has been enabled by ℓ was already enabled in Ψ'. Then by Proposition 2.3 (1), the invariant is also satisfied.

Otherwise, the invariant ensures that the transition $Q \xrightarrow{\ell}$ is enabled after Ψ' as well. Note that by performing the transition after Ψ', the same syntactic transformation occurs (which may be the reduction of a redex, the addition of a new redex by input, or the suppression of an action by failing reduction). Then both the claim and the invariant are still satisfied. We complete Ψ by reducing any linear redex remaining. □

The proof of Lemma 5.7 is now easy.

Proof of Lemma 5.7 Let $\langle s, S\rangle \in \mathsf{WFT}(P)$, where $s = \widehat{\Phi}$ and $S = \mathtt{blocked}(\Phi)$, for some fair and finitely failing sequence of transitions Φ. Then by Lemma H.1, there is a fair and finitely failing sequence Ψ of molecular actions from P such that $\langle \widehat{\Psi}, \mathtt{blocked}(\Psi)\rangle = \langle s, S\rangle$. Then $\langle s, S\rangle \in \mathsf{WFT}(Q)$. The other direction is immediate, by using Lemma H.1 on both sides. □

This lemma applies directly to a cas-queue process and a lock-queue one related by $\mathcal{R}_{\mathsf{lock,cas}}$, where a failure in the former is mapped to a failure *before* lock acquisition in the latter.

Proof of Lemma 5.8 We only need to consider the subset of $\mathsf{WFT}(\mathsf{CQemp}(r))$ which is generated by molecular action sequences (Lemma 5.7). Then we just need to play the simulation game between the two queues. Note that $\mathsf{LQemp}(r)\,\mathcal{R}_{\mathsf{lock,cas}}\mathsf{CQemp}(r)$. Throughout the simulation, we satisfy the invariant that the cas-queue process is in general form (which is ensured by our usage of molecular actions that allow us to derive only threads in LMF) and that the lock-queue process is in normal form (which we ensure by normalisation), up to $\equiv$.

Let Φ be a finitely failing and fair sequence of molecular actions from $\mathsf{CQemp}(r)$. We simulate Φ from $\mathsf{LQemp}(r)$ as follows. When an internal, non-commit and non-failing transition occurs in Φ, we do not do anything. Each time a visible or commit molecular action occurs in Φ, we let the lock-queue process do the same transition (which must be enabled, by Lemma G.27 (1) and (2)) and then we normalise the thread in which the transition occurred by applying Lemma G.25. Since in the normalisation we only performed τ-transitions, the two processes obtained are still related (by definition of $\mathcal{R}_{\mathsf{lock}}$). Each time a failing reduction occurs in some g-thread of Φ, we do a failing reduction in the g-thread of Ψ. This is possible because the corresponding thread is in LMF: since the lock-queue process is in normal form, the g-thread is either a pending process (in which case the commit fails) or an answer (in which case the answer fails). Note also that after a failing reduction, the thread where it occurred is reduced to some sub-process $P \equiv \mathbf{0}$. Then the cas-queue and the lock-queue processes are still in general form and in normal form up to $\equiv$, respectively.

Let Ψ be the transition sequence obtained by this simulation. Clearly, Φ and Ψ have the same sequence of visible transitions. Moreover, each failure in some g-thread of Φ was mapped to a failure in the g-thread of Ψ. By Proposition 5.1, such a failure blocks only g in Φ. And the same can be said for Ψ, since the failure occurs *outside* the critical section (the proof is analogous to that of Proposition 5.1). Then $\langle \widehat{\Phi}, \mathtt{blocked}(\Phi)\rangle = \langle \widehat{\Psi}, \mathtt{blocked}(\Psi)\rangle \in \mathsf{WFT}(\mathsf{LQemp}(r))$. □

As we already mentioned in § 4, we only need to show a weak fair finitely failing trace of $\mathsf{LQemp}(r)$ not belonging to $\mathsf{CQemp}(r)$. We give the proof once again, augmented with all the details:

Proof of Theorem 5.9 By Proposition 4.14 and by Proposition G.28, we have established the bisimilarity results:

$$\mathsf{AQ}(r, \langle \emptyset,\, \varepsilon,\, \emptyset\rangle) \approx \mathsf{LQemp}(r) \approx \mathsf{CQemp}(r)$$

By Lemma 5.8, we have established the inclusion:

$$\mathsf{WFT}(\mathsf{CQemp}(r)) \subseteq \mathsf{WFT}(\mathsf{LQemp}(r))$$

Then we only need to show its strictness.

Suppose from $\mathsf{LQemp}(r)$ we receive a (enqueue or dequeue) request. Then we do the following reduction and commit action and we perform a failing reduction before releasing the lock. Then we receive infinitely many other requests, generating threads which we make progress up to the pending process. Such a sequence is finitely failing and it is also fair: in particular it is maximal, because no thread can progress further (from the second thread on, all are stuck before commit).

By contradiction, assume that a fair and finitely failing transition sequence Φ with the same visible behaviour (reception of infinitely many inputs) and the same set of blocked outputs (all) is possible from $\mathsf{CQemp}(r)$. By Proposition 5.1, for each blocked output, Φ must contain a failing reduction. Then Φ contains infinitely many failing reductions. Contradiction. □

I Supplementary Operational Arguments

I.1 Scheduling Strategy for the Proof of Lemma F.1

We define a deterministic and fair derivation strategy on typed processes, where non-linear inputs never occur. Then by applying this strategy on $\Gamma \vdash P$ and then repeatedly on its derivatives, we eventually obtain a transition sequence which satisfies all the requirements.

We start from *decorating processes with counters*, so that, at each step, we perform the transition which has waited the longest. Non-linear input transitions are associated to an ω timestamp. Concretely, for all natural numbers k, we define a function $time_k$ on processes as follows:

$$\begin{aligned}
time_k(u\&^{L}_{i\in I}\{l_i(\vec{x}_i).P_i\}) &= \langle u\&^{L}_{i\in I}\{l_i(\vec{x}_i).P_i\}, k\rangle \\
time_k(u\&^{\mathrm{M}}_{i\in I}\{l_i(\vec{x}_i).P_i\}) &= \langle u\&^{\mathrm{M}}_{i\in I}\{l_i(\vec{x}_i).P_i\}, \omega\rangle \\
&\qquad (\mathrm{M} \neq L) \\
time_k(\overline{u} \oplus l\langle\vec{e}\rangle) &= \langle \overline{u} \oplus l\langle\vec{e}\rangle, k\rangle \\
time_k(\texttt{if } \langle e, o\rangle \texttt{ then } P \texttt{ else } Q) &= \langle \texttt{if } \langle e, o\rangle \texttt{ then } P \texttt{ else } Q, k\rangle \\
time_k(P|Q) &= time_k(P)|time_{k'}(Q) \\
&\qquad (\ k' = max(time_k(P)) + 1\ ) \\
time_k((\boldsymbol{\nu}\, u)P) &= (\boldsymbol{\nu}\, u)time_k(P) \\
time_k((\mu X(\vec{x}).P)\langle\vec{e}\rangle) &= (\mu X(\vec{x}).time_k(P))\langle\vec{e}\rangle \\
time_k(X\langle\vec{x}\rangle) &= X\langle\vec{x}\rangle \\
time_k(\mathbf{0}) &= \mathbf{0}
\end{aligned}$$

where the function $max(time_k(P))$ returns the highest finite timestamp which appears in $time_k(P)$. In the above definitions we indicated a modality annotation only for those processes where the modality would discriminate the function. Elsewhere we avoided such an annotation, implicitly assuming that the function definition applies to all modalities. We could have avoided modality annotations everywhere, but we would have had to give the definitions directly on typed processes. Instead, we decided to give a decoration to typed processes starting from the decoration on untyped ones, through the function $time$:

$$time(\Gamma \vdash P) \;=\; Gamma \vdash time_0(P)$$

In general we write P^T for a process P decorated with unique timestamps (that is, unique up to ω). Also, given a process with timestamps P^T such that $\Gamma \vdash P$, we may write $\Gamma \vdash P^T$ to indicate its typed version.

We are now ready to define the derivation strategy on processes with timestamps. Consider a typed process with timestamps $\Gamma \vdash P^T$. In order to decide which transition to perform, we first spot the branching, selection or conditional with the lowest timestamp in P^T. Then:

- if it is a linear branching with subject c and c has linear input type in Γ, we perform this linear input transition;
- if it is a linear branching with subject c but c does not have a linear input type in Γ, it means that the input has already occurred and the branching is waiting to be synchronised with a complementary selection. Then the input may not be done (Γ does not allow it). Instead we spot the next lowest timestamp in P and we proceed with the case analysis on its associated action;
- if it is a selection with subject c such that the corresponding output transition on c is allowed by Γ, we perform this output transition;
- if it is a selection with subject c whose synchronisation with the complementary branching on c is enabled in P, then we reduce this synchronisation;
- if it is a selection with subject c, for which none of the above two cases applies, it means that the complementary branching is not ready for synchronisation, then its reduction is not enabled. Then we just skip it, we consider the next lowest timestamp in P and we do the case analysis on its associated action;
- if it is a conditional, we reduce it.

Now assume that we performed the following transition (shown here in the undecorated setting):

$$\Gamma \vdash P \xrightarrow{\ell} \Delta \vdash Q$$

since we want to keep track of time, we may not decorate $\Delta \vdash Q$ in the naive way, i.e. through the function $time$ again. The reason is that the transition may have generated new redexes, which we want to reduce after the old ones. Then we define $Q^{T'}$ as follows:

- on any branching, selection or conditional which has not been reduced, we keep the same timestamp it had in P^T;
- on the sub-process generated by the previous transition, we apply $time_k$, where $k = max(P^T) + 1$.

Then we apply the same strategy on $\Delta \vdash Q^{T'}$, recursively.

The key property of the above strategy is that timestamps are assigned incrementally, so that they are unique and the next transition is always selected among those which have been waiting the most. Note however that the transition that is performed does not always correspond to the lowest timestamp. But as we have explained, if this does not happen it means that the transition associated to the lowest timestamp *may not* be performed. Moreover, according to the current strategy, the same transition shall be performed as soon as it may. Then fairness is satisfied, since at any point only a finite number of transitions may be performed.

Note also that no non-linear input transitions are performed, since non-linear branchings are associated to ω timestamps, so that they are never reduced except in internal synchronisations (i.e. when the complementary selection is selected).

Finally, maximality is achieved by the recursive application of the strategy.