Type-Based Liveness Guarantee in the Presence of Nontermination and Nondeterminism

Nobuko Yoshida

Department of Computing, Imperial College London, UK

Abstract. This paper investigates a type-based framework to guarantee a basic liveness property in the π-calculus. The resulting liveness property ensures that the action at a specified channel will eventually fire, in spite of the presence of nondeterminism and possibly diverging computation. We first integrate nontermination into the linear π-calculus introduced in [33], for which we prove the liveness by a translation into the linear π-calculus, preserving a specific sequence of typed actions. We then extend the calculus with first-order state, and prove the liveness property via a translation into the linear π-calculus with nondeterministic sums. The systematic method based on techniques from term rewriting systems and analysis of operational structures associated with linearity leads to a clean proof of the liveness. The liveness property is interesting not only in its own right, but also in its nontrivial semantic consequences, including decidability of equations and non-interference theorems of secure functional and imperative programming languages.

1 Introduction

In the distributed systems, the correctness of concurrent programs or systems has been often classified using two terms: the *safety* and *liveness* properties [4, 21, 23]. Intuitively a safety property asserts that nothing bad happens during execution of programs, while a liveness property asserts that something good eventually happens. It is well-known that the typing systems of the programming languages can offer a static verification technique to guarantee safety properties by means of the subject reduction. In the sequential computation, an important and typical liveness property is that programs or systems eventually terminate regardless of the evaluational strategies. In this context, if we have programs translatable into the simply typed λ-calculus, both safety and liveness are automatically ensured by static type-checking. In concurrent and distributed computation, however, termination of the entire system is too strong requirement. In multi-threaded nondeterministic languages such as Java [18], one may wish to get an answer at some particular port without going into a loop, while assuming some part contains nonterminating behaviour. Here liveness can be formalised as whatever state a system is in, it is always the case that an action at the specified channel will eventually be executed [4, 23]. If we can ensure such an interaction-based liveness property by type-checking, we can verify components in isolation, and build up a larger system by local composition of smaller subsystems one-by-one, without having to verify a strong property (such as liveness) for an entire system. This benefit of the type-based approach suggests that it is worthwhile to investigate a uniform framework of typing systems which can ensure both liveness and safety properties where nonterminating and nondeterministic computations coexist.

The π-calculus [12, 26] is a formalism of computing in which we can compositionally represent diverse behaviours, from sequential, deterministic computation to those which involve nontermination and nondeterminism, by decomposing them into a single communication primitive, the name passing. In our preceding work, we introduced basic type structures which precisely characterise two prominent classes of pure functional computation: the class of terminating functions and that of possibly diverging ones, which we call *linear* [33] and *affine* [5]. The class of types crucially relies on duality, and are closely related with Game Semantics [3, 15, 17] and Linear Logic [8]. We make the meaning of these terms more precise below:

- *Affinity.* This denotes possibly diverging behaviour in which a question is given an answer at most once.

- *Linearity.* This denotes terminating behaviour in which a question is always given an answer precisely once.

As given above, "linearity" means more than just terminating behaviour: it indicates, in essence, a process always returns an answer if it is asked for. As a simple example, $!b(c).\overline{c}\langle 1\rangle$ in the standard π-calculus syntax is a process which, when asked at b, surely answers at c with 1; so c is linear in this process. The mixture of nontermination (affinity) and nondeterminism (state) with linearity is fine-grained: even inside a process engaged in a linear interaction, nonterminating computation may take place, and linear interactions themselves can involve nondeterministic computation (e.g. by access to shared variables). For example, take the following process:

$$!b(xc).(\overline{x}\langle v\rangle \mid \overline{c}\langle w\rangle) \mid (\boldsymbol{\nu}\, ac)(\overline{b}\langle ac\rangle \mid !a(x).\overline{a}\langle x\rangle \mid c(y).P)$$

This process has first an interaction at b, then does an infinite series of actions at a while having a linear answer at c: in the above terminology, it is affine at a, while it is linear at c. This fine-grained mixture is vital for flexible embedding of various programming constructs [13, 16], hence a proof of liveness needs a careful analysis of causality of name-passing. The paper shows that uniform and consistent type-based integration of affinity and state into linearity is possible, and that we can establish liveness properties in the integrated calculi with a rigorous proof method based on traditional techniques from term rewriting systems and analysis of operational structures of the π-calculus.

The integrated typed π-calculus can faithfully embed a multi-threaded imperative language with first-order references and remote procedure calls, with local liveness property. Thus, the results obtained in the calculus can in principle be transferred to such programming languages (cf.[13, 16]). Not only the liveness property is interesting in its own right, but also it has nontrivial semantic consequences, including decidability of equations and non-interference theorems of secure functional and imperative programming languages such as DCC [1] and Smith-Volpano languages [30]. We briefly discuss a couple of such applications at the end.

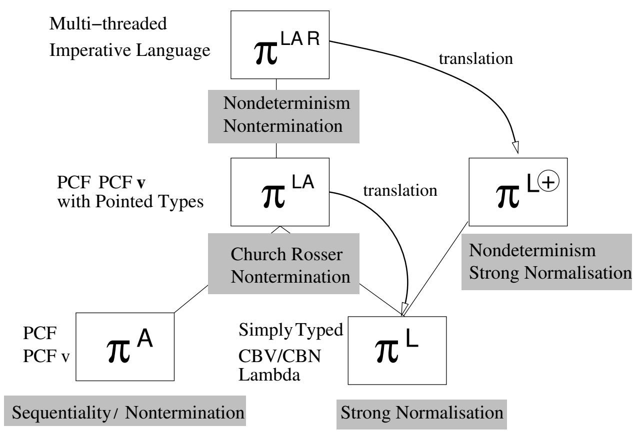

Fig. 1. A Family of Linear/Affine π-Calculi

A picture of typed calculi used in this paper is given in Figure 1. Each box lists the typed π-calculus ("L", "A", "R" and "$\oplus$" mean linear, affine, first-order reference, and summation respectively). The left-hand side of the box shows systems we can embed in the calculus. It also shows the relationship of translations from the basic calculi to target calculi for proving the liveness property. The grey box shows a basic property satisfied by the calculus.

Related Work Since [21], the terms *safety* and *liveness* play an important role in the analysis of concurrent programs and distributed systems. In the π-calculus, various versions of linear types have been developed, and some of them (e.g.[10, 11, 19, 28, 31]) are used to avoid certain kinds of communication deadlock. Among them, Yoshida [31] introduced a typing system in which actions on linear channels fire following the order of graph types. Sangiorgi proposed a typing system to guarantee a property called receptiveness where an input prefix is always available [27]. He also recently proposed a strong normalising typing system with summation and the first order recursive agents explicitly adding a global constraint of name ordering [29]. Kobayashi proposed a typing system to ensure lock-freedom where CCS-terms with time tags are used as types [19]. None of them treats the liveness property involving a non-trivial use of replication; for example, in the linear-affine calculus, interaction restricted on the liveness channels can still embed both *semantics* and *dynamics* of the simply typed lambda-calculus. Our method to construct an integrated calculus combining restricted calculi in a bottom-up way offers a clear articulation of communication behaviours, supported by full abstraction results of the known calculi/languages [5, 6, 33, 34]. This feature, among others, leads to a clear proof methodology for the liveness, as well as for the applications to secrecy in programming languages, which are not found in the previous work. In another paper [16], we adapt our framework to this practical direction.

Apart from the application to secrecy, since our approach based on type-checking does not limit to finite state systems, a possible interesting research direction is to use the typed π-calculus as a precise computational representation of Alpen and Schneider's liveness and safety in the linear/branching times [24]. This may offer an integration of logics and type disciplines to develop and extend tools for the distributed systems based on process calculi (e.g. [20, 23]).

Outline of the Paper Section 2 integrates affinity (nontermination) into linearity. Section 3 formalises and proves the liveness property of the linear/affine calculus. Section 4 further integrates state, which gives a nondeterministic calculus, and proves its liveness property. Section 5 discusses a couple of applications of the liveness property. The full technical development of the presented material is found in [32].

Acknowledgements The author thanks the anonymous referees for their useful technical comments. She also thanks Jim Laird for discussion on his proof method in [22]. The results in Section 5.1 are formulated based on a suggestion from Jeff Magee. She also thanks Martin Berger and Kohei Honda for her on-going collaboration on typed π-calculi and their applications. She is partially supported by EPSRC grant GR/R33465/01.

2 Integrating Linearity and Nontermination

2.1 Processes

Following [5, 13, 33, 34], we use the asynchronous version of the π-calculus [12] with bound output [27].[1] Let $x, y, \ldots, a, b, \ldots$ range over a countable set of names (also called channels). The set of untyped terms, *processes*, is given by the following grammar.

$$
\begin{array}{llll}
P ::= x(\vec{y}).P \text{ input} & \mid\ P \mid Q \quad \text{parallel} & \mid\ \mathbf{0} & \text{inaction} \\
\quad\ \mid\ \overline{x}(\vec{y})\,P \text{ output} & \mid\ (\boldsymbol{\nu}\, x)P \text{ hiding} & \mid\ !x(\vec{y}).P & \text{replication}
\end{array}
$$

The bound/free names are defined as usual. We assume names in a vector $\vec{y}$ are pairwise distinct. Up to the structural equality, whose definition is given in [5, 33, 34], the output $\overline{x}(\vec{y})Q$ acts as $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|Q)$ in the standard syntax. The structural equality $\equiv$ is given as in [5, 33, 34] (cf. [34, App.A]).

[1] We can use free output with precisely the same results [33]. Bound output however gives simpler proofs for many results, including liveness properties, cf. [5, 6, 27, 33].

The reduction is generated by the following rules, closing under parallel composition, restriction and output, taking processes modulo $\equiv$.

$$
\begin{array}{rcl}
x(\vec{y}).P \mid \overline{x}(\vec{y})\,Q & \longrightarrow & (\boldsymbol{\nu}\,\vec{y})(P \mid Q) \\
!x(\vec{y}).P \mid \overline{x}(\vec{y})\,Q & \longrightarrow & !x(\vec{y}).P \mid (\boldsymbol{\nu}\,\vec{y})(P \mid Q),
\end{array}
$$

The multi-step reduction $\twoheadrightarrow$ is given as $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^*$. As simple examples, a *copy-cat agent* [17] is given as $[x \to y] \stackrel{\text{def}}{=} !x(c).\overline{y}(c')c'.\overline{c}$, which links two locations x and y; then an *omega agent* is defined as $\Omega_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\,y)([u \to y] | [y \to u])$, which immediately diverges when it is interrogated at u.

2.2 Integrating Linearity and Affinity

In [5] and [33], we introduced affine and linear typing systems for the above syntax. There are the following three key steps for a coherent integrations of affinity and linearity:

- distinction between linear and affine action modes;
- constraint in channel types (sorting [25]) such that linear output cannot be carried by affine replication; and
- constraint in typing such that affine input never suppresses linear output.

In the following we introduce these elements one by one.

Step 1: Action Modes. Action modes [5, 33] specify modes of interaction at each channel. We integrate action modes from [5, 33], distinguishing linear modes and affine modes.

	Input mode		Output mode		Uncomposable mode
$\downarrow_{\text{L}}$	Linear input	$\uparrow_{\text{L}}$	Linear output	$*_{\text{L}}$	Linear uncomposable
$!_{\text{L}}$	Linear server	$?_{\text{L}}$	Client request to $!_{\text{L}}$		
$\downarrow_{\text{A}}$	Affine input	$\uparrow_{\text{A}}$	Affine output	$*_{\text{A}}$	Affine uncomposable
$!_{\text{A}}$	Affine server	$?_{\text{A}}$	Client request to $!_{\text{A}}$		

"L" modes are linear modes from [33], while "A" modes are affine modes given in [5]. We leave detailed illustration of these concepts to Appendix A. The difference between linearity and affinity is that, in a linear channel, an interaction takes place *precisely once*, while it does so *at most once* in an affine channel. A subtle point is that we need to keep this distinction even for replicated modes (see Step 2 below). $p, p', \ldots$ range over action modes. p_{I} (resp. p_{O}) denotes input (resp. output) modes. The pair of input/output modes in each row are *dual* to each other, writing $\overline{p}$ for the dual of p. We also set $\mathcal{M}_{\downarrow} = \{\downarrow_{\text{L}}, \downarrow_{\text{A}}\}$, $\mathcal{M}_{\uparrow} = \{\uparrow_{\text{L}}, \uparrow_{\text{A}}\}$, $\mathcal{M}_{!} = \{!_{\text{L}}, !_{\text{A}}\}$, $\mathcal{M}_{?} = \{?_{\text{L}}, ?_{\text{A}}\}$ and $\mathcal{M}_{*} = \{*_{\text{L}}, *_{\text{A}}\}$.

Step 2: Channel Types. The syntax of *channel types* are given by:

$$\tau \ ::= \ (\vec{\tau})^p \ \mid \ *_{\text{L}} \ \mid *_{\text{A}}$$

$\overline{\tau}$, the *dual* of τ, dualises all action modes: for example $\overline{((*_{\text{L}})^{\downarrow_{\text{L}}})^{?_{\text{L}}}} = ((*_{\text{L}})^{\uparrow_{\text{L}}})^{!_{\text{L}}}$. The *mode* of types is defined by $\mathsf{md}((\vec{\tau})^p) = p$, $\mathsf{md}(*_{\text{L}}) = *_{\text{L}}$, and $\mathsf{md}(*_{\text{A}}) = *_{\text{A}}$. The partial operation $\tau_1 \odot \tau_2$ is defined following [5, 33], see Appendix B.

A key element for integration is the following constraints on channel types.

(C1) In $(\vec{\tau})^p$ with $p \in \mathcal{M}_{\downarrow}$, $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for each i. Dually when $p \in \mathcal{M}_{\uparrow}$.
(C2) In $(\vec{\tau})^{!_{\text{L}}}$ $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for each i except one j for which $\mathsf{md}(\tau_j) \in \mathcal{M}_{\uparrow}$. Dually for $(\vec{\tau})^{?_{\text{L}}}$ (for simplicity we often set $j = n$).
(C3) In $(\vec{\tau})^{!_{\text{A}}}$, $\mathsf{md}(\tau_i) \in \mathcal{M}_{?}$ for each i except at most one j for which $\mathsf{md}(\tau_j) = \uparrow_{\text{A}}$. Dually for $(\vec{\tau})^{?_{\text{A}}}$.

These constraints come from [5, 33] except for (C3), which allows $\uparrow_{\mathrm{L}}$-type to be carried only by a linear replication $!_{\mathrm{L}}$. This gives a minimum separation between affinity and linearity. We illustrate its need by a simple example. Take $\Omega_x \stackrel{\text{def}}{=} (\boldsymbol{\nu} y)(!x(c).\overline{y}(c')c'.\overline{c} \mid !y(c').\overline{x}(c)c.\overline{c'})$. Because computation at x and y do not terminate, x, y should have mode $!_{\mathrm{A}}/?_{\mathrm{A}}$. If we type c, c' with $\downarrow_{\mathrm{L}}/\uparrow_{\mathrm{L}}$ (contradicting C3), then $\overline{x}(c)c.P$ can never receive an input at c, i.e. we lose linearity at c. This is because we do not care about termination in affine replication. Note also this constraint assumes the distinction between $!_{\mathrm{L}}$ and $!_{\mathrm{A}}$ as action modes.

Step 3: Typing. An *action type* is an assignment of channel types to names [5, 33] together with causality information, represented as a directed graph with each node being a name-channel type pair. A directed edge represents causality among channels, which is always either from $!_{\mathrm{L}}$ to $?_{\mathrm{L}}$ or from $\downarrow_{\mathrm{L}}$ to $\uparrow_{\mathrm{L}}$ following [33]. $A, B, \ldots$ range over action types. $\mathsf{md}(A)$ and $\mathsf{fn}(A)$ denote the sets of modes and names in A, respectively. Detailed illustration of action types and the partial operations $A \odot B$ and $A \asymp B$ are given in Appendix B.

The typing rules are given in Figure 2, using the sequent of form $\vdash P \triangleright A$. In Figure 2, we use the following notations.

- $A^{\vec{y}:\vec{\tau}}$ — A in which each $y_i : \tau_i$ in $\vec{y} : \vec{\tau}$ occurs.
- $\vec{p}A$ $\;(?A)$ — A such that $\mathsf{md}(A) \subset \{\vec{p}\}$ $\;(\mathsf{md}(A) \subset \mathcal{M}_?)$
- $A^{\text{-}x}$ — A such that $x \notin \mathsf{fn}(A)$.
- $A/\vec{x}$ — the result of taking off $x_i : \tau_i$ from A.
- $A \otimes B$ — a disjoint union of A and B s.t. $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$.
- $x : \tau \to A$ — Add $x : \tau$ to A with an edge from $x : \tau$ to A's active nodes.

In the last line we say a node is *active* if it has no incoming edges. We say P *is typable under* A if $\vdash P \triangleright A$ is derivable from these rules.

Leaving detailed illustration of each rule to Appendix C, we focus on the key constraint for consistent integration in $(\mathsf{In}^{\downarrow_{\mathrm{A}}})$. In this rule, (apart from the same constraint coming from affine typing in [5]) we do *not* allow $\downarrow_{\mathrm{A}}$ to suppress $\uparrow_{\mathrm{L}}$. As an example, if $x : (\vec{\tau})^{\downarrow_{\mathrm{A}}}$ suppresses $y : (\vec{\tau}')^{\uparrow_{\mathrm{L}}}$, because the dual of $\downarrow_{\mathrm{A}}$ at x may not arrive (as in $x.\overline{y} \mid y.\overline{x}$), the output at y may not fire, violating the linearity at y. Other than this, the typing rules allow maximum mixing among linear/affine interaction. Simple examples of typing follow (see Appendix C for the details). Below we let $\mathtt{unit}^{\circ} \stackrel{\text{def}}{=} (()^{\uparrow_{\mathrm{L}}})^{!_{\mathrm{L}}}$ and $\mathtt{unit}^{\bullet} = (()^{\uparrow_{\mathrm{A}}})^{!_{\mathrm{A}}}$. These types respectively specify a unit linear (resp. affine) server which necessarily (resp. possibly) returns a fixed answer. $[x \to y]$ and Ω_x appeared in 2.1.

$$
\text{(Zero)}\ \frac{-}{\vdash \mathbf{0} \triangleright \_}
\qquad
\text{(Par)}\ \frac{\vdash P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2}{\vdash P_1|P_2 \triangleright A_1 \odot A_2}
\qquad
\text{(Res)}\ \frac{\vdash P \triangleright A^{x:\tau} \qquad \mathsf{md}(\tau) \in \mathcal{M}_{*,!}}{\vdash (\boldsymbol{\nu}\, x:\tau)P \triangleright A/x}
\qquad
\text{(Weak)}\ \frac{\vdash P \triangleright A^{\text{-}x} \qquad \mathsf{md}(\tau) \in \mathcal{M}_{*,?}}{\vdash P \triangleright A \otimes x:\tau}
$$

$$
(\mathsf{In}^{\downarrow_{\mathrm{L}}})\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes \uparrow_{\mathrm{L}} A^{\text{-}x} \otimes \uparrow_{\mathrm{A}} B^{\text{-}x} \otimes ?C^{\text{-}x}}{\vdash x(\vec{y}:\vec{\tau}).P \triangleright (x:(\vec{\tau})^{\downarrow_{\mathrm{L}}} \to A) \otimes B \otimes C}
\qquad
(\mathsf{In}^{!_{\mathrm{L}}})\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes ?_{\mathrm{L}} A^{\text{-}x} \otimes ?_{\mathrm{A}} B^{\text{-}x}}{\vdash !\, x(\vec{y}:\vec{\tau}).P \triangleright (x:(\vec{\tau})^{!_{\mathrm{L}}} \to A) \otimes B}
$$

$$
(\mathsf{In}^{\downarrow_{\mathrm{A}}})\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes \uparrow_{\mathrm{A}} ?B^{\text{-}x}}{\vdash x(\vec{y}:\vec{\tau}).P \triangleright x:(\vec{\tau})^{\downarrow_{\mathrm{A}}} \otimes B}
\qquad
(\mathsf{In}^{!_{\mathrm{A}}})\ \frac{\vdash P \triangleright \vec{y}:\vec{\tau} \otimes ?A^{\text{-}x}}{\vdash !\, x(\vec{y}:\vec{\tau}).P \triangleright x:(\vec{\tau})^{!_{\mathrm{A}}} \otimes A}
\qquad
\text{(Out)}\ \frac{\vdash P \triangleright A^{\vec{y}:\vec{\tau}} \quad A/\vec{y} \asymp x:(\vec{\tau})^{p\circ}}{\vdash \overline{x}(\vec{y})P \triangleright A/\vec{y} \odot x:(\vec{\tau})^{p\circ}}
$$

Fig. 2. Linear/Affine Typing Rules

- (constants) $\vdash !u(c).\overline{c} \triangleright x : \mathtt{unit}^{\circ}$.
- (copy-cat) $\vdash [x \to y] \triangleright x : \mathtt{unit}^{\circ} \to y : \overline{\mathtt{unit}^{\circ}}$, $\vdash [x \to y] \triangleright x : \mathtt{unit}^{\bullet} \otimes y : \overline{\mathtt{unit}^{\bullet}}$, and $\vdash [x \to y] \triangleright x : \mathtt{unit}^{\bullet} \otimes y : \overline{\mathtt{unit}^{\circ}}$, but $\not\vdash [x \to y] \triangleright x : \mathtt{unit}^{\circ} \otimes y : \overline{\mathtt{unit}^{\bullet}}$.
- (omega) Let $\vdash \Omega_x \triangleright x : \mathtt{unit}^{\bullet}$ is typable, while $\vdash \Omega_x \triangleright x : \mathtt{unit}^{\circ}$ is not typable.
- An example from Section 1: $\vdash \overline{b}(ac)(\Omega_a^{\mathtt{unit}^{\bullet}} \mid c.\overline{e}) \triangleright b : (\mathtt{unit}^{\bullet}()^{\downarrow_{\mathsf{L}}})^{?_{\mathsf{L}}} \otimes e : ()^{\uparrow_{\mathsf{L}}}$. Here we can expect the answer at e surely arrives.

2.3 Syntactic Properties of Linear/Affine Typing

We denote the calculus by π^{LA}. π^{LA} is conservative extensions of the original affine and linear calculi [5, 33], which we hereafter write π^{L} (resp. π^{A}), respectively. We write $\vdash_{\mathsf{L}} P \triangleright A$ (resp. $\vdash_{\mathsf{A}} P \triangleright A$) if P is inferred only with linear (resp. affine) types, i.e. in π^{L} (resp. π^{A}). We list elementary properties of π^{LA}. The proof is straightforward from the same results in [5, 33].

Proposition 1.

1. (inclusion) $\vdash_{\mathsf{L}} P \triangleright A$ *implies* $\vdash P \triangleright A$. *Also* $\vdash_{\mathsf{A}} P \triangleright A$ *implies* $\vdash P \triangleright A$.
2. (subject reduction) *If* $\vdash P \triangleright A$ *and* $P \twoheadrightarrow Q$ *then we have* $\vdash Q \triangleright A$.

The π^{LA}-calculus also satisfies the one-step diamond property of $\longrightarrow$ [5, 33]. The following lemma says the typing rules ensure global partial order between free linear names [2] by local type-checking.

Lemma 1. (acyclicity between linear names) *Assume* $\vdash P \triangleright A$. $G(P)$ *denotes a directed graph s.t.: (1) nodes are linear names, i.e.* $\{a \mid a \in \mathsf{fn}(P) \ \wedge\ \mathsf{md}(A(a)) \in \{\uparrow_{\mathsf{L}}, \downarrow_{\mathsf{L}}, *_{\mathsf{L}}, ?_{\mathsf{L}}, !_{\mathsf{L}}\}\}$*; and (2) edges are given by:* $x \mapsto y$ *iff* $P \equiv (\boldsymbol{\nu}\,\vec{z})(Q|R)$ *such that* $Q \equiv x(\vec{w}).Q_0$ *or* $Q \equiv !x(\vec{w}).Q_0$ *where* y *occurs free in* Q_0, $x \notin \{\vec{z}\}$ *and* $y \notin \{\vec{z}\vec{w}\}$. *A* cycle *in* $G(P)$ *is a sequence of form* $x \mapsto y_1 ... \mapsto y_n \mapsto x$ $(n \geq 0)$ *with* $y_i \neq x$. *Then* $G(P)$ *has no cycle.*

For example, let $P \equiv [x \to y] \mid [y \to z]$. Then $G(P) = x \mapsto y \mapsto z$. Note also that we permit a cycle between affine names (e.g. in Ω^x).

3 Liveness in Affine/Linear System

3.1 Liveness

First we define liveness at a linear channel. The liveness means that we can eventually get a unique reply at a linear channel.

Definition 1. (linear liveness) Write $P \Downarrow_x$ when $P \twoheadrightarrow \overline{x}(\vec{y})P'$ for some $\vec{y}$ and P'. Then we say P *eventually answers at* x iff whenever $P \twoheadrightarrow Q$ we have $Q \Downarrow_x$.

Note that even if P eventually answers at x, it may not guarantee termination of P (for example, $P \equiv \overline{x} \mid \Omega_a$ diverges, while $P \Downarrow_x$). Now we state the first main theorem. Below, we say A is *closed* if $\mathsf{md}(A) \subseteq \mathcal{M}_{*,!}$. Closedness of A corresponds to closed terms of the λ-calculus: a process with this type is composed with all opponent (complementary) processes.

Theorem 1. (liveness in π^{LA}) *Suppose* $\vdash P \triangleright A \otimes x : \tau$ *with* A *closed and* $\mathsf{md}(\tau) = \uparrow_{\mathsf{L}}$. *Then* P *eventually answers at* x.

The theorem says that if a typed agent waits for an answer at a truly linear channel x, then the answer is guaranteed to arrive at x eventually, even with co-existence of non-terminated computation. To prove liveness is a difficult task due to complex intermediate interaction sequences before reaching an output at x. As a suggestion of difficulty, take a process:

$$P \equiv \overline{z}(y_1)y_1.\overline{z}(y_2)y_2...\overline{z}(y_n)y_n.\overline{x}$$

which has type $z : \overline{\mathtt{unit}^{\circ}} \otimes x : ()^{\uparrow_{\mathrm{L}}}$. This process calls a resource $!z(y).Q$ with type $z : \overline{\mathtt{unit}^{\circ}}$ a number of times. Then $P \mid !z(y).Q$ has type $z : \mathtt{unit}^{\circ} \otimes x : ()^{\uparrow_{\mathrm{L}}}$ where $\mathtt{unit}^{\circ}$ is closed. However, we cannot guarantee the liveness at x until we can maintain the liveness to each y_i from Q in which y may occur in the deep inside of possibly nonterminating prefixes. For example, if P is composed with Ω_z in § 2, then (since y_i does not fire) x never comes out. However, using the type structure of π^{LA}, we can carry out a rigorous operational analysis of linear actions, centring on the structure called *linear call-sequences* [34]. Once we define this structure, we construct a simple translation from π^{LA} to π^{L} so that it preserves linear call-sequences. The rest is the standard technique for terminating known in term rewriting systems where we use not only rewriting (reduction) but also labelled actions.

3.2 Linear Call-Sequences

In order to prove Theorem 1, we need to generalise Theorem 1 where A in the definition may not be closed: *if P has a linear output type at x, there always exist a finite shortest chain of linear transitions to reach x without being interfered by nonterminating actions.* To define such a chain precisely, we use the labelled transition relations. The actions $l, l', \ldots$ are given by the grammar:

$$l \ ::= \ \boldsymbol{\tau} \mid x(\vec{y}) \mid \overline{x}(\vec{y}).$$

$t, t', \ldots$ range over sequences of labels. $\mathsf{bn}(l)$ denotes bound names in l. If $l \neq \boldsymbol{\tau}$, we write $\mathsf{sbj}(l)$ for the initial free name of l. Given A, we often write l^{τ} if $A(\mathsf{sbj}(l)) = \tau$ and define $\mathsf{md}(l^{\tau}) = \mathsf{md}(\tau)$. We often write l^{p} if $\mathsf{md}(l^{\tau}) = p$. We define $\overline{x(\vec{y})} = \overline{x}(\vec{y})$ and $\overline{\overline{x}(\vec{y})} = x(\vec{y})$ ($\overline{\boldsymbol{\tau}}$ undefined). Using these labels, the typed transition $P^{A} \xrightarrow{l} Q^{B}$ is defined in Appendix D.

Define a set of *linear types* Lin as the smallest set generated by: $(\tau_1 ... \tau_n)^{p} \in \mathsf{Lin}$ if (1) $p \in \{\uparrow_{\mathrm{L}}, \downarrow_{\mathrm{L}}\}$ or (2) $\exists i.\mathsf{md}(\tau_i) \in \{\uparrow_{\mathrm{L}}, \downarrow_{\mathrm{L}}\}$. Note that $(\tau_1 ... \tau_n)^{?_{\mathrm{L}}}$ (resp. $(\tau_1 ... \tau_n)^{!_{\mathrm{L}}}$) is *not* in Lin if it carries an affine channel, i.e. $\mathsf{md}(\tau_n) = \downarrow_{\mathrm{A}}$ (resp. $\uparrow_{\mathrm{A}}$). Set $\curvearrowright = \curvearrowright_{\mathsf{b}} \cup \curvearrowright_{\mathsf{p}}$ where $l \curvearrowright_{\mathsf{b}} l'$ means that the subject of l' is bound by l, and $l \curvearrowright_{\mathsf{p}} l'$ means that l' is under input prefix l. We write $(l,\ \bar{l}) \curvearrowright l_2$ if $P \stackrel{l_2}{\Longrightarrow} Q$ and P has subterms Q_1 and Q_2 such that $Q_1 \xrightarrow{l} Q_1'$ and $Q_2 \stackrel{\bar{l} \cdot l_2}{\Longrightarrow} Q_2'$ with $\bar{l} \curvearrowright l_2$; similarly we define $l_1 \curvearrowright (l,\ \bar{l})$. We often denote $(l,\ \bar{l})$ by $\boldsymbol{\tau}$, and extend the above chain $l_1 \curvearrowright \boldsymbol{\tau}^{*} \curvearrowright l_2$ and denote it $l_1 \curvearrowright^{+} l_2$ ([5, Appendix F] gives a detailed definition using occurrences of terms). We can now define a linear chain of actions reaching a final answer.

Definition 2. (linear call-sequence) A *linear call-sequence (l.c.s.) to l under A* is a (possibly infinite) sequence of actions with type Lin whose visible actions have the following shape:

$$(l_0^{\downarrow_{\mathrm{L}}} \curvearrowright^{+})\ l_1^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_2^{\downarrow_{\mathrm{L}}} \curvearrowright^{+} l_3^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_4^{\downarrow_{\mathrm{L}}} \curvearrowright^{+} \cdots l_{2n-1}^{?_{\mathrm{L}}} \curvearrowright_{\mathsf{b}} l_{2n}^{\downarrow_{\mathrm{L}}} \cdots$$

where $\tau_i, \tau_{ij}, \tau_{ij} \in \mathsf{Lin}$ for all $l_i^{\tau_i} \curvearrowright^{*} (l_{ij}^{\tau_{ij}}, \bar{l}_{ij}) \curvearrowright^{*} l_{i+1}^{\tau_{i+1}}$ in the above sequence, and if it is finite, we have $l_{2n}^{\downarrow_{\mathrm{L}}} \curvearrowright_{\mathsf{p}} l^{\uparrow_{\mathrm{L}}}$ for some $n \lneq \omega$.

As an example of l.c.s, let $P \equiv \overline{a}(x)x.\overline{b}(y)y.\overline{e} \,|\, [b \to a]$. Then l.c.s. to e is $\overline{a}(x) \cdot x \cdot (\overline{b}(y), b(y)) \cdot (\overline{y}, y) \cdot \overline{e}$. We have the following main lemma for l.c.s. ($|A|$ is the set of nodes in A). The proof follows that of [34, Lemma 2], hence omitted.

Lemma 2. (l.c.s.) *Suppose $\vdash P \triangleright A$ with $x : (\vec{\tau})^{\uparrow_{\mathrm{L}}} \in |A|$. Then there is a (possibly infinite) l.c.s. $l_1 \cdot l_2 \cdots l_n \cdots$ to l s.t. $\mathsf{sbj}(l) = x$ and $P \stackrel{l_1 \cdots l_n \cdots}{\longrightarrow}$.*

This lemma also means if $\neg P \Downarrow_x$, then there should be an infinite l.c.s. from P. Note that the ordering of names appeared in l.c.s. does *not* follow $G(P)$ in Lemma 1 (see the example before the proposition where $G(P) = b \mapsto a$). Hence it is difficult to directly define the well-founded ordering on linear call sequences.

3.3 Translation from π^{LA} to π^{L}

We shall prove finiteness of l.c.s. on π^{LA} in Lemma 2 via a simple translation of π^{LA} into π^{L}. This is defined by eliminating all of the subterms which inductively contain affine inputs/outputs (which are never visited in linear call sequences).

Definition 3. The mappings $[\![\tau]\!]$, $[\![A]\!]$, $[\![l]\!]$ and $[\![P]\!]_{|A|} : \pi^{\mathsf{LA}} \to \pi^{\mathsf{L}}$ (for $\vdash P \triangleright A$) are inductively defined as in Figure 3.

Proposition 2. *1.* $[\![\tau]\!]$ *only includes types in* π^{L}.
2. $\vdash P \triangleright A$ *implies* $\vdash_{\mathsf{L}} [\![P]\!]_{|A|} \triangleright [\![A]\!]$.
3. $P^A \xrightarrow{l} Q^B$ *implies* $[\![P]\!]_{|A|} \xrightarrow{[\![l]\!]} [\![Q]\!]_{|B|}$. *Also* $[\![A(\mathsf{sbj}(l))]\!] \neq \varepsilon$ *implies* $[\![l]\!] \neq \varepsilon$.

(1) is by induction on $[\![\tau]\!]$, while (2) and (3) are by rule induction on $\vdash P \triangleright A$ and $P^A \xrightarrow{l} Q^B$, respectively.

(Channel Type)

$$[\![(\tau_1...\tau_n)^p]\!] = \begin{cases} \varepsilon & ((\tau_1...\tau_n)^p \notin \mathsf{Lin}) \\ ([\![\tau_1]\!]...[\![\tau_n]\!])^p & ((\tau_1...\tau_n)^p \in \mathsf{Lin}) \end{cases} \qquad \begin{array}{l} [\![*_{\mathsf{A}}]\!] = \varepsilon \\ [\![*_{\mathsf{L}}]\!] = *_{\mathsf{L}} \end{array}$$

(Action Type) Below we set (1) $[\![\tau]\!] = \varepsilon$ (2) $[\![\tau]\!] \neq \varepsilon$.

$$[\![\emptyset]\!] = \emptyset \qquad [\![a : \tau \to A]\!] = \begin{cases} [\![A]\!] & (1) \\ a : [\![\tau]\!] \to [\![A]\!] & (2) \end{cases} \qquad [\![a : \tau \otimes A]\!] = \begin{cases} [\![A]\!] & (1) \\ a : [\![\tau]\!] \otimes [\![A]\!] & (2) \end{cases}$$

(Label) (1) $[\![\tau]\!] = \varepsilon$ (2) $[\![\tau]\!] \neq \varepsilon$ with $\tau = (\vec{\tau}\vec{\tau}')^p$, $[\![\tau_i]\!] = \varepsilon$, $[\![\tau_i']\!] \neq \varepsilon$.

$$[\![a(\vec{x}\vec{y})^\tau]\!] = \begin{cases} \varepsilon & (1) \\ a(\vec{y}) & (2) \end{cases} \qquad [\![\overline{a}(\vec{x}\vec{y})^\tau]\!] = \begin{cases} \varepsilon & (1) \\ \overline{a}(\vec{y}) & (2) \end{cases} \qquad [\![(l^\tau, \overline{l}^{\overline{\tau}})]\!] = \begin{cases} \varepsilon & (1) \\ ([\![l^\tau]\!], [\![\overline{l}^{\overline{\tau}}]\!]) & (2) \end{cases}$$

(Term) (1) $[\![A(a)]\!] = \emptyset$ (2) $[\![A(a)]\!] \neq \emptyset$ with $A(a) = (\vec{\tau}\vec{\tau}')^p$, $[\![\tau_i]\!] = \varepsilon$, $[\![\tau_i']\!] \neq \varepsilon$.

$$[\![\mathbf{0}]\!]_{|A|} \stackrel{\text{def}}{=} \mathbf{0} \qquad [\![P_1 \mid P_2]\!]_{|A|} \stackrel{\text{def}}{=} [\![P_1]\!]_{|B_1|} \mid [\![P_2]\!]_{|B_2|} \quad (\vdash P_i \triangleright B_i,\ B_1 \odot B_2 = A)$$

$$[\![a(\vec{x} : \vec{\tau}, \vec{y} : \vec{\tau}').P]\!]_{|A|} = \begin{cases} \mathbf{0} & (1) \\ a(\vec{y} : [\![\vec{\tau}']\!]).[\![P]\!]_{|A| \cup \vec{x}:\vec{\tau} \cup \vec{y}:\vec{\tau}'} & (2) \end{cases}$$

$$[\![!\, a(\vec{x} : \vec{\tau}, \vec{y} : \vec{\tau}').P]\!]_{|A|} = \begin{cases} \mathbf{0} & (1) \\ !\, a(\vec{y} : [\![\vec{\tau}']\!]).[\![P]\!]_{|A| \cup \vec{x}:\vec{\tau} \cup \vec{y}:\vec{\tau}'} & (2) \end{cases}$$

$$[\![\overline{a}(\vec{x} : \vec{\tau}, \vec{y} : \vec{\tau}')P]\!]_{|A|} = \begin{cases} \mathbf{0} & (1) \\ \overline{a}(\vec{y} : [\![\vec{\tau}']\!])[\![P]\!]_{|A| \cup \vec{x}:\vec{\tau} \cup \vec{y}:\vec{\tau}'} & (2) \end{cases}$$

$$[\![(\boldsymbol{\nu}\, a : \tau)P]\!]_{|A|} = \begin{cases} [\![P]\!]_{|A| \cup a:\tau} & [\![\tau]\!] = \varepsilon \\ (\boldsymbol{\nu}\, a : [\![\tau]\!])[\![P]\!]_{|A| \cup a:\tau} & [\![\tau]\!] \neq \varepsilon \end{cases}$$

Fig. 3. Translation from π^{LA} into π^{L}

Now the proof of Theorem 1 is by a simple contradiction. Remember if $\vdash_{\mathsf{L}} P \triangleright A$, then the call sequences which start from P are always finite [34, Lemma 2] due to the strong normalisation of π^{L}.

Proof of Theorem 1. Suppose $\vdash P \triangleright A \otimes x : \tau$ with A closed and $\mathsf{md}(\tau) = \uparrow_{\mathsf{L}}$. By Lemma 2, there is a possibly infinite linear call sequence $l_1 \cdot l_2 \cdots$ to x. Suppose $l_1 \cdot l_2 \cdots$ is infinite, i.e. $P \equiv P_0 \xrightarrow{l_1} P_1 \xrightarrow{l_2} P_3 \cdots$. Then by Prop. 2(3), we have $[\![P]\!] \equiv [\![P_0]\!] \xrightarrow{[\![l_1]\!]} [\![P_1]\!] \xrightarrow{[\![l_2]\!]} [\![P_3]\!] \cdots$ with $[\![l_i]\!] \neq \varepsilon$, which however contradicts [34, Lemma 2]. Now suppose $P \twoheadrightarrow P'$. By Prop. 1(2), we have $\vdash P' \triangleright A \otimes x : \tau$. Since A is closed, the l.c.s. is a single output action at x, as required. ■

4 Nontermination, State and Liveness

4.1 Reference Agents

In the following we further enrich π^{LA} with stateful computation, using a simple constant agent representing (first-order) references. This makes the resulting calculus truly nondeterministic, in the sense that Church-Rosser property no longer holds. In spite of this, the proof method used in the previous section can be smoothly extended, first by introducing a non-deterministic linear π-calculus, $\pi^{\mathsf{L}\oplus}$, and by proving its strong normalisation using a term rewriting technique; and, secondly, by translating the stateful calculus into $\pi^{\mathsf{L}\oplus}$ preserving linear call-sequences (this proof method essentially follows [22]).

We consider reference agents which hold boolean values for brevity. Together with boolean references, we introduce branching and selection [5, 16, 33, 34].

$$
\begin{array}{llll}
P ::= \ldots \mid \mathsf{ref}\langle x\mathbb{T}\rangle \mid \mathsf{ref}\langle x\mathbb{F}\rangle & \text{reference} & \mid \overline{x}\mathtt{in}_i(\vec{z})Q & \text{selection} \\
\quad\quad\;\; \mid x[\&_i(\vec{y_i}).P_i] & \text{branch} & \mid !x[\&_i(\vec{y_i}).P_i] & \text{replication} \\
\tau ::= \ldots \mid [\&_i\vec{\tau_i}]^{p_\mathtt{I}} \mid [\otimes_i\vec{\tau_i}]^{p_\mathtt{O}} & & &
\end{array}
$$

Branching and selection correspond to sums in the λ-calculus and additives in Linear Logic [8]. We assume the branching has the finite index. Their interaction is defined as follows.

$$x[\&_i(\vec{y_i}).P_i] \mid \overline{x}\mathtt{in}_j(\vec{y_j})\,Q \longrightarrow (\boldsymbol{\nu}\,\vec{y_j})(P_j \mid Q)$$

Similarly for the replication. $\mathsf{ref}\langle x\mathbb{T}\rangle$ and $\mathsf{ref}\langle x\mathbb{F}\rangle$ are introduced as constants, which we call *reference agents*. Their dynamics is given as follows ($\mathbb{B}$ means either $\mathbb{T}$ or $\mathbb{F}$).

$$
\begin{array}{ll}
(\mathtt{read}) & \mathsf{ref}\langle x\mathbb{T}\rangle \mid \overline{x}\mathtt{in}_1(c)P \longrightarrow \mathsf{ref}\langle x\mathbb{T}\rangle \mid (\boldsymbol{\nu}\,c)(P \mid \overline{c}\mathtt{in}_1) \\
 & \mathsf{ref}\langle x\mathbb{F}\rangle \mid \overline{x}\mathtt{in}_1(c)P \longrightarrow \mathsf{ref}\langle x\mathbb{F}\rangle \mid (\boldsymbol{\nu}\,c)(P \mid \overline{c}\mathtt{in}_2) \\
(\mathtt{write}\mathbb{T}) & \mathsf{ref}\langle x\mathbb{B}\rangle \mid \overline{x}\mathtt{in}_2(c)P \longrightarrow \mathsf{ref}\langle x\mathbb{T}\rangle \mid (\boldsymbol{\nu}\,c)(P \mid \overline{c}) \\
(\mathtt{write}\mathbb{F}) & \mathsf{ref}\langle x\mathbb{B}\rangle \mid \overline{x}\mathtt{in}_3(c)P \longrightarrow \mathsf{ref}\langle x\mathbb{F}\rangle \mid (\boldsymbol{\nu}\,c)(P \mid \overline{c})
\end{array}
$$

It has three branches, of which the left one corresponds to the "read" option, while the middle and third ones correspond to the "write true" and "write false" options. For information, we also present the recursive definition of $\mathsf{ref}\langle xy\rangle$.

$$
\begin{array}{l}
\mathsf{ref}\langle x\mathbb{T}\rangle \stackrel{\text{def}}{=} x[(c).(\mathsf{ref}\langle x\mathbb{T}\rangle \mid \overline{c}\mathtt{in}_1)\ \&\ (c).(\mathsf{ref}\langle x\mathbb{T}\rangle \mid \overline{c})\ \&\ (c).(\mathsf{ref}\langle x\mathbb{F}\rangle \mid \overline{c})] \\
\mathsf{ref}\langle x\mathbb{F}\rangle \stackrel{\text{def}}{=} x[(c).(\mathsf{ref}\langle x\mathbb{F}\rangle \mid \overline{c}\mathtt{in}_2)\ \&\ (c).(\mathsf{ref}\langle x\mathbb{T}\rangle \mid \overline{c})\ \&\ (c).(\mathsf{ref}\langle x\mathbb{F}\rangle \mid \overline{c})]
\end{array}
$$

$[\&_i\vec{\tau_i}]^{p_\mathtt{I}}$ and $[\otimes_i\vec{\tau_i}]^{p_\mathtt{O}}$ are called branching and selection types, and they are dual to each other. The conditions (C1-C3,G1-G2) in § 2 are kept as they are. We often use $\mathbb{B}^\circ \stackrel{\text{def}}{=} [\ \otimes\ ]^{\uparrow_\mathsf{L}}$ (boolean type) and $\mathtt{ref}^\circ \stackrel{\text{def}}{=} [\mathbb{B}^\circ\&()^{\uparrow_\mathsf{L}}\&()^{\uparrow_\mathsf{L}}]^{!_\mathsf{L}}$. The typing rules for π^{LAR} are all of the original rules in Figure 2, together with the additional rules given in Figure 4.

We call this calculus π^{LAR}. We can easily check Proposition 1 and Lemma 1 hold in π^{LAR}. As is well-known, combination of state and concurrency leads to a loss of Church-Rosser property by interference, as the following example shows:

$$\mathsf{ref}\langle x\mathbb{F}\rangle \mid \overline{x}\mathtt{in}_1(c)c[.\overline{y}\mathtt{in}_1\ \&\ .\overline{y}\mathtt{in}_2] \mid \overline{x}\mathtt{in}_2(e)e.\overline{f}$$

where the output at y differs according to whether the "read" action reaches x first or not. In spite of this, we have the following (the notion "eventually answers" in π^{LAR} is defined incorporating selection as in Definition 1).

$$\text{(Ref-t)} \quad \vdash \ \mathsf{ref}\langle x\mathbb{T}\rangle \triangleright x : \mathtt{ref}^{\circ} \qquad \text{(Ref-f)} \quad \vdash \ \mathsf{ref}\langle x\mathbb{F}\rangle \triangleright x : \mathtt{ref}^{\circ}$$

$$\text{(Bra}^{\downarrow_{\mathrm{A}}}\text{)}\ \frac{\vdash P_i \triangleright \vec{y}_i : \vec{\tau}_i \otimes \uparrow_{\mathrm{A}} ?\, A^{\text{-}x}}{\vdash x[\&_i(\vec{y}_i).P_i] \triangleright x : [\&_i\vec{\tau}_i]^{\downarrow_{\mathrm{A}}} \otimes A} \qquad \text{(Sel)}\ \frac{\vdash P \triangleright A^{\vec{y}:\vec{\tau}_j} \quad A/\vec{y} \asymp x : [\otimes_i \vec{\tau}_i]^{p\circ}}{\vdash \overline{x}\mathtt{in}_j(\vec{y})P \triangleright A/\vec{y} \odot x : [\otimes \vec{\tau}_i]^{p\circ}}$$

(Bra$^{\downarrow_{\mathrm{L}}}$), (Bra$^{!_{\mathrm{L}}}$), and (Bra$^{!_{\mathrm{A}}}$) are defined as in Figure 2.

Fig. 4. Typing Rules for Reference, Branching and Selection

Theorem 2. (liveness in π^{LAR}) *Suppose* $\vdash P \triangleright A \otimes x : \tau$ *with* A *closed and* $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$ *in* π^{LAR}. *Then* P *eventually answers at* x.

We use the rest of the section for the proof of Theorem 2. First we note the following property (note l.c.s. can be infinite; the definition for l.c.s. is the same as Definition 2).

Lemma 3. *Suppose* $\vdash P \triangleright A$ *in* π^{LAR} *with* $x : \tau \in |A|$ *and* $\mathsf{md}(\tau) = \uparrow_{\mathrm{L}}$. *Then there is a (possibly infinite) l.c.s.* $l_1 \cdot l_2 \cdots l_n \cdots$ *to* l *s.t.* $\mathsf{sbj}(l) = x$ *and* $P \stackrel{l_1 \cdot l_2 \cdots l_n \cdots}{\longrightarrow}$.

4.2 Linear π-Calculus with Internal Sums

We translate the linear call sequences of the π^{LAR} to the linear π-calculus with internal sums [22], called $\pi^{\mathsf{L}\oplus}$. The syntax of the $\pi^{\mathsf{L}\oplus}$-calculus, as well as one of the reduction relations we use, follows.

$$P ::= \ \dots \ | \ \oplus_i P_i \qquad\qquad \oplus_i P_i \ \longrightarrow_{\oplus} \ P_i$$

Obviously $\longrightarrow_{\oplus}$ is strong normalising. The additional typing rule is given as:

$$\text{(Sum)}\ \frac{\vdash P_i \triangleright ?_{\mathrm{L}}A \otimes \uparrow_{\mathrm{L}}B}{\vdash \oplus_i \ P_i \triangleright A \otimes B}$$

where B has empty or a singleton (cf. App C). To prove the termination of $\pi^{\mathsf{L}\oplus}$, we use a basic result from TRS — termination of the union of term-rewriting systems is reducible to the termination of each under a constraint.

Lemma 4. (commutation [7]) *Let* R_1 *and* R_2 *be two strongly normalising TRSs with two reduction relations* $\longrightarrow_1$ *and* $\longrightarrow_2$*, respectively. If* $\longrightarrow_1$ quasi commutes *over* $\longrightarrow_2$*, i.e. if for any* $P \longrightarrow_1 Q \longrightarrow_2 R$ *in* $R_1 \cup R_2$*, then there exists* Q' *such that* $P \longrightarrow_2 Q' \twoheadrightarrow_1 R$*, then the union* $R_1 \cup R_2$ *is strong normalising.*

$(\oplus, \ \longrightarrow_{\oplus})$ does not commute over $(\pi^{\mathsf{L}}, \ \longrightarrow)$. However, it does so over the extended reduction $\mapsto$ introduced in [5,6,33], which contracts all redexes under bodies. We leave the definition of $\mapsto$ to Appendix E. Thus we obtain the following. Below $\vdash_{\mathrm{L}\oplus} P \triangleright A$ means P is inferred by the typing rules in π^{L} and (Sum) above.

Proposition 3. (strong normalisation of $\pi^{\mathsf{L}\oplus}$) *If* $\vdash_{\mathrm{L}\oplus} P \triangleright A$*, then* P *is strong normalising w.r.t. the reduction* $\mapsto \cup \longrightarrow_{\oplus}$.

4.3 Proof of Theorem 2

We can now define translations from π^{LAR} to $\pi^{\mathsf{L}\oplus}$ by adding the following map for the reference agents to those in Figure 3 (the mapping of branching/selection types are given following Figure 3 in the obvious way).

$$[\![\mathsf{ref}\langle x\mathbb{T}\rangle]\!]_{|A|} \stackrel{\text{def}}{=} [\![\mathsf{ref}\langle x\mathbb{F}\rangle]\!]_{|A|} \stackrel{\text{def}}{=} !x[(c).(\overline{c}\mathtt{in}_1 \oplus \overline{c}\mathtt{in}_2) \ \& \ (c).\overline{c} \ \& \ (c).\overline{c}]$$

The idea of this mapping is to keep a track of all possibilities (truth or false) as a summand of name passing trees until the linear answer is actually used.

Proposition 4. *1.* $[\![\tau]\!]$ *only includes types in* π^{L}.
2. $\vdash P \triangleright A$ *implies* $\vdash_{\mathsf{L}\oplus} [\![P]\!]_{|A|} \triangleright [\![A]\!]$.
3. $P^A \stackrel{l}{\longrightarrow} Q^B$ *implies* $[\![P]\!]_{|A|} \stackrel{[\![l]\!]}{\Longrightarrow} [\![Q]\!]_{|B|}$. *Also* $[\![A(\mathsf{sbj}(l))]\!] \neq \varepsilon$ *implies* $[\![l]\!] \neq \varepsilon$.

Using Propositions 3 and 4 instead of [34, Lemma 2] and Proposition 2, respectively, as well as via Lemma 3, we have now established Theorem 2.

5 Applications: Bisimilarities and Secrecy

5.1 Decidability of Actions and Equations

Linear liveness (Theorems 1 and 2) is by itself a useful property: for example, via translation, we can ensure an answer to an invocation in programming languages with local and remote procedures is returned eventually. Apart from this direct interest, linear liveness has a few significant applications. In open distributed systems, one may wish to check whether processes P and Q return the same answer at a particular port, ignoring other ports. Write $\vdash P \Downarrow^i_x$ when there exists R such that $P \twoheadrightarrow \overline{x}\mathtt{in}_i(\vec{z})R$ where $\vdash P \triangleright A \otimes x : [\oplus\vec{\tau}_i]^{\uparrow\mathsf{L}}$ with A closed. Intuitively $P \Downarrow^i_x$ says (for a linear x) the value i is observable at channel x. By finiteness of the linear call sequences, as well as the decidability of $P_1 \equiv P_2$ and the finiteness of branching index, we know:

Proposition 5. (decidability) *Suppose* $\vdash P \triangleright A \otimes x : [\oplus\vec{\tau}_i]^{\uparrow\mathsf{L}}$ *with* A *closed in the* π^{LA} *or* π^{LAR}*-calculus. Then* $P \Downarrow^i_x$ *is decidable.*

This behavioural property is further extended to equations, leading to finite axiomatisation of the weak bisimilarity over linear behaviour in π^{LA}: we denote $\approx^{\circ}_{\mathtt{L}}$ for the standard typed weak bisimilarity [5, 33, 34] except if $P \approx^{\circ}_{\mathtt{L}} Q$, then P and Q only simulate linear actions each other, i.e. $\mathbf{R}$ is a linear bisimilar, if $\vdash P\mathbf{R}Q \triangleright A$ implies whenever $P \stackrel{l^{\tau}}{\longrightarrow} P'$ with $\tau \in \mathsf{Lin}$, then there is a transition sequence $Q \stackrel{\hat{l}}{\Longrightarrow} Q'$ such that $\stackrel{\hat{l}}{\Longrightarrow} \stackrel{\text{def}}{=} \stackrel{l_1 \ldots l_n}{\longrightarrow}$ with $\tau_i \in \mathsf{Lin}$ for all $l_i^{\tau_i}$, and $\vdash P\mathbf{R}Q \triangleright A'$ (as well as the symmetric case). By first cutting all affine terms by $[\![\ ]\!]$ in Figure 3,[2] then applying the same method in [33, Section 4], we immediately obtain:

Proposition 6. (finite axiomatisation) $\vdash P \approx^{\circ}_{\mathtt{L}} Q \triangleright A$ *iff* $[\![P]\!]_{|A|} \longleftrightarrow_{\mathsf{L}} [\![Q]\!]_{|A|}$ *where* $\mapsto_{\mathsf{L}}$ *is the extended reduction on* π^{L} *(Appendix E),* $\longleftrightarrow_{\mathsf{L}}$ *is the transitive, symmetric closure of* $\mapsto_{\mathsf{L}} \cup \equiv$.

This result means that equivalence of two processes in π^{LA} are finitely checkable in the infinite state computation, as far as we do not care interaction on non-terminating channels. Finally we note π^{LAR} has non-determinism, so that we cannot use the extended reduction to obtain the axioms. We leave a similar finite axiomatisation for π^{LAR} open.

5.2 Linear/Affine Bisimilarity and Secrecy

In [34], we presented a theory of bisimilarity for π^{L} in which we abstract away not only $\boldsymbol{\tau}$-actions but also non-$\boldsymbol{\tau}$ actions which do not affect well-typed observers. This bisimilarity equates a larger set of processes than those by the standard typed bisimilarity. For example, using the translation in [16, 33], we can obtain the following equation only using labelled transition of the π-calculus.

$$\mathtt{if}\ w\ \mathtt{then}\ x := \mathbb{T}\ \mathtt{else}\ x := \mathbb{T} \quad \cong \quad x := \mathbb{T}$$

We sketch below how the same framework is systematically extendible to π^{LA} and π^{LAR} via the liveness property (existence of such an extension is briefly mentioned, but not technically developed, in [34]).

[2] The procedure $[\![P]\!]_{|A|}$ obviously terminates finitely.

To define such a bisimilarity, we first classify the typed actions in π^{LA} and π^{LAR} by the following two criteria: whether typable actions affect the environment non-trivially; and whether these actions are guaranteed to take place. Hereafter we decompose $?_{\mathrm{L}}$ into read ($?_{\mathrm{L}}$) and write ($?_{\mathrm{L}\mu}$), and refine (Ref) rules, precisely following [16].

Definition 4. (affecting and enabled types)

1. τ is *affecting* iff there exist $\vdash P_{1,2} \triangleright x : \tau$ and a typed context $C[\cdot]$ such that $\vdash C[P_i] \triangleright u : ()^{\uparrow_{\mathrm{A}}}$, $C[P_i] \Downarrow_u$ and $\neg\, C[P_j] \Downarrow_u$ with $i \neq j$.
2. τ is *enabling* iff $\vdash P \triangleright x : \tau$ implies $P \longrightarrow^* P' \xrightarrow{l}$ such that $\mathsf{sbj}(l) = x$. τ is *enabled* if $\overline{\tau}$ is enabling.

As an illustration, the process $\overline{c}\mathtt{in}_2$ with c being a linear channel, emits 2 via c, hence (the type of) c is affecting. Similarly, Ω_x and $!x(c).\overline{c}$ are typable under the same type (with c affine), but divergence/convergence at c makes their information content different, so c is affecting. Also $?_{\mathrm{L}}$ is not affecting, while $?_{\mathrm{L}\mu}$ is so due to the "write" action in references agents. Table 1 summarises how typed actions are classified in π^{LA} and π^{LAR} (there are some exceptions such as $()^{\uparrow_{\mathrm{L}}}$). $()$ means unary, and $\&/\oplus$ means branching/selection.

	τ	$\downarrow_{\mathrm{L}}()$	$\uparrow_{\mathrm{L}}()$	$\downarrow_{\mathrm{L}}\&$	$\uparrow_{\mathrm{L}}\oplus$	$\downarrow_{\mathrm{A}}$	$\uparrow_{\mathrm{A}}$	$!_{\mathrm{L}}$	$?_{\mathrm{L}}$	$?_{\mathrm{L}\mu}$	$!_{\mathrm{A}}$	$?_{\mathrm{A}}$
affecting	no	no	yes	no	yes	no	yes	yes	no	yes	yes	no
enabled	yes	yes	yes	yes	yes	no	yes	no	yes	yes	no	yes

Table 1. Classification of Actions

The linear bisimilarities are defined as [34, Definition 6] (we denote $\approx_{\mathsf{LA}}$ and $\approx_{\mathsf{LAR}}$ for π^{LA} and π^{LAR}, respectively), whose congruency can be proved by the liveness of the linear call sequences, following [34, Section 4]. The induced bisimilarity can justify many interesting equations over linear/affine π-terms. As a further application, we may develop a behavioural theory of secure information flow for the π-calculus as in [34], which uses a secrecy-sensitive bisimilarity built on the top of $\approx_{\mathsf{LA}}$ and $\approx_{\mathsf{LAR}}$, precisely following [34, Definitions 7-8]. This theory gives short proofs for the properties of type-based secrecy π-calculi in [16], as well as ensuring secrecy for a larger set of processes. For example, assuming $\bot$ and $\top$ are low and high secrecy level and command C is safe, the theory can justify the safety of the following program, which is not type-checkable in [30].

$$\mathtt{if}\ w^{\top}\ \mathtt{then}\ x^{\bot} := \mathbb{T}; C\ \mathtt{else}\ x^{\bot} := \mathbb{T}; C \quad (\cong\ x^{\bot} := \mathbb{T}; C)$$

Here we can assume C contains nonterminating behaviour, e.g. `while` loop. The secrecy theory is also directly applicable to such programming languages as the call-by-name and call-by-value DCC [1] (via π^{LA}) and multi-threaded Smith-Volpano imperative language [30] extended with higher-order procedures constructs (via π^{LAR}) along the line of [16]. Among others, noninterference results in these languages are direct consequences of those in the secure version of π^{LA} and π^{LAR}. Details of these applications will be presented elsewhere.

References

1. Abadi, M., Banerjee, A., Heintze, N. and Riecke, J., A core calculus of dependency, *POPL'99*, ACM, 1999.
2. Abramsky, S., Computational interpretation of linear logic. *TCS*, Vol. 111 (1993) 3–57, 1993.
3. Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF. *Info. & Comp.* 163 (2000), 409–470.
4. Alpen, B. and Schneider, F.B, Defining liveness, *Info. Processing Letter*, 21(4):181–185, Oct, 1985.
5. Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, *TLCA'01*, LNCS 2044, 29–45, Springer, 2001. The full version available at www.dcs.qmw.ac.uk/~kohei.

6. Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus. To appear in *FoSSaCs'03* (ETAPS'03), LNCS, Springer, 2003.
7. Dershowitz, N. and Jouannaud, J.-P., Rewriting Systems, Handbook of TCS, Vol. B, Chap. 6, 243–320, MIT, 1990.
8. Girard, J.-Y., Linear Logic, *TCS*, 50, 1–102, 1987.
9. Hennessy, M. and Riely, J., Information flow vs resource access in the asynchronous pi-calculus, *ICALP'00*, LNCS 1853, 415–427, Springer, 2000. The full version appeared in *TOPLAS*.
10. Honda, K., Composing Processes, *POPL'96*, 344–357, ACM, 1996.
11. Honda, K., Kubo, M. and Vasconcelos, V., Language Primitives and Type Discipline for Structured Communication-Based Programming. *ESOP'98*, LNCS 1381, 122–138. Springer-Verlag, 1998.
12. Honda, K. and Tokoro, M. An object calculus for asynchronous communication. *ECOOP'91*, LNCS 512, 133–147, 1991.
13. Honda, K., Vasconcelos, V. and Yoshida, N., Secure Information Flow as Typed Process Behaviour, *ESOP'00*, LNCS 1782, 180–199, 2000.
14. Honda, K. and Yoshida, N. On Reduction-Based Process Semantics. *TCS*, 151, 437–486, 1995.
15. Honda, K. and Yoshida, N., Game-Theoretic Analysis of Call-by-Value Computation. *TCS*, 221 (1999), 393–456, 1999.
16. Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL'02*, pp.81–92, ACM, 2002. The full version as DoC technical report 2002/13, Imperial College London, 2002.
17. Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285–408, 2000.
18. Java, Sun Microsystems Inc., http://www.javasoft.com/.
19. Kobayashi, N., A type system for lock-free processes, *I. & C.*, 2001.
20. Holzmann, G.J., The model checker SPIN. IEEE Transactions on Software Engineering SE-23, 5 (May), 279–295.
21. Lamport, L., Proving the correctness of multiprocess programs, *IEEE Transactions on Software Engineering SE-3*, 2:125–143, Mar, 1977.
22. Laird, J., A deconstruction of non-deterministic classical cut elimination, *TLCA'01*, LNCS 2044, 268–282, Springer, 2001.
23. Magee, J. and Kramar, J., Concurrency: state models & Java programs, John Wiley & Sons, 1999.
24. Manolios, P. and Trefler, R., Safety and Liveness in Branching Time, *LICS'01*, 366–374, IEEE, 2001.
25. Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.
26. Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Info. & Comp.* 100(1), pp.1–77, 1992.
27. Sangiorgi, D. π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.
28. Sangiorgi, D., The name discipline of uniform receptiveness, *TCS*, 221, 457–493, 1999.
29. Sangiorgi, D., Termination of processes, a draft, December, 2001.
30. Smith, G. and Volpano, D., Secure information flow in a multi-threaded imperative language, 355–364, *POPL'98*, ACM, 1998.
31. Yoshida, N. Graph Types for Mobile Processes. *FST/TCS'16*, LNCS 1180, 371–386, Springer, 1996.
32. Yoshida, N. The full version of this paper, MCS techincal report, 2002/20, University of Leicester, 2002.
33. Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *LICS'01*, 311–322, IEEE, 2001. The full version as DoC technical report 2003/1, Imperial College London, 2003.
34. Yoshida, N., Honda, K. and Berger, M., Linearity and Bisimulation, *FoSSaCs'02* (ETAPS'02), LNCS, Springer, 2002. A full version as MCS technical report, 2001-48, University of Leicester, 2001. Available at www.mcs.le.ac.uk/˜nyoshida/paper.html.

A Action Modes

– "$\downarrow$" mode is associated with an input (e.g. x in $x(y).P$).
– "!" mode is associated with a replicated input (e.g. x in $!x(y).P$).
– "$\uparrow$" (resp. "?") mode is associated with an output delivered to "$\downarrow$" (resp. "!").

For example, if $\overline{x}(\vec{y})P$ is composed with $x(y).P$, then x in $\overline{x}(\vec{y})P$ has a $\uparrow$-mode, and if it is composed with $!x(y).P$, then x has a ?-mode.

We also use the modes "$*_{\mathsf{L}}$" and "$*_{\mathsf{A}}$" which mean uncomposability of linear and affine channels; for example, if $x.\mathbf{0}$ has a $\downarrow_{\mathsf{L}}$-mode and $\overline{x}$ has a $\uparrow_{\mathsf{L}}$-mode, $(x.\mathbf{0} \mid \overline{x})$ has $*_{\mathsf{L}}$-mode at x, which is uncomposable with any process which has x, hence $*_{\mathsf{L}}$ can guarantee the linearity of channel x (similarly for $*_{\mathsf{A}}$).

B Composition of Channel/Action Types

Channel Types $\odot$ is the least commutative partial operation on channel types such that:

$$\begin{array}{lllll}
(1) & \tau \odot \overline{\tau} = *_{\mathsf{L}} & (\mathsf{md}(\tau) = \downarrow_{\mathsf{L}}) & \tau \odot \overline{\tau} = *_{\mathsf{A}} & (\mathsf{md}(\tau) = \downarrow_{\mathsf{A}}) \\
(2) & \tau \odot \tau = \tau & (\mathsf{md}(\tau) \in \mathcal{M}_{?}) & \tau \odot \overline{\tau} = \tau & (\mathsf{md}(\tau) \in \mathcal{M}_{!})
\end{array}$$

Intuitively, (1) says once we compose input-output linear channels it becomes uncomposable. (2) says that a server should be unique, to which an arbitrary number of clients can request interactions. For example, $!x.\mathbf{0} \mid !x.\mathbf{0}$ is never typed because of $()^{!_{\mathsf{L}}} \not\asymp ()^{!_{\mathsf{L}}}$, while $\overline{x} \mid \overline{x}$ is typable by $()^{?_{\mathsf{L}}}$ at x, and $!x.\mathbf{0} \mid \overline{x} \mid \overline{x}$ is so by $()^{!_{\mathsf{L}}}$ at x. Note also the composition between affine and linear types is prohibited; for example, $()^{\downarrow_{\mathsf{L}}} \odot ()^{\uparrow_{\mathsf{A}}}$ is undefined, while by (1), $()^{\downarrow_{\mathsf{L}}} \odot ()^{\uparrow_{\mathsf{L}}} = *_{\mathsf{L}}$. Similarly $(()^{\uparrow_{\mathsf{L}}})^{!_{\mathsf{L}}} \odot (()^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}}$ is undefined, while by (2), $(()^{\uparrow_{\mathsf{A}}})^{!_{\mathsf{A}}} \odot (()^{\downarrow_{\mathsf{A}}})^{?_{\mathsf{A}}} = (()^{\uparrow_{\mathsf{A}}})^{!_{\mathsf{A}}}$. If $\tau \odot \tau'$ is defined we say they *compose*.

Action Types An action type is a finite directed acyclic graph such that:

(G1) Each node has form $x : \tau$. No two nodes in the graph have identical subjects.
(G2) Each edge has form $x : \tau \to x' : \tau'$ with either $\mathsf{md}(\tau) = \downarrow_{\mathsf{L}}$ and $\mathsf{md}(\tau') = \uparrow_{\mathsf{L}}$, or $\mathsf{md}(\tau) = \mathbf{!}_{\mathsf{L}}$ and $\mathsf{md}(\tau') = \mathbf{?}_{\mathsf{L}}$.

Note, in **(G2)** we do not record dependency for affine channels. This is because we permit circularity, hence divergence, for affine channels (but not for linear channels). $A, B, \ldots$ range over action types. We can also describe A in the form of grammar by duplicating ?-nodes [33, Example 1] as follows.

$$A ::= \emptyset \mid a : \tau \mid A \otimes B \mid a : \tau \to (b : \tau_1 \otimes b : \tau_2 \cdots \otimes b : \tau_n)$$

The partial operator $A \odot B$ is defined iff channel types in common names compose by $\odot$ above, and, moreover, the adjoined graph do not have a cycle. If so, the result is a graph in which, in the adjoined graph, each maximal causal chain is collapsed into an edge connecting its two ends (see [33, §.2] or [34, App. B] for further details). To avoid divergence at linear channels, this operator ensures that processes never exhibit circular dependency in actions. For example, $x : \tau_1 \to y : \tau_2$ and $y : \overline{\tau_2} \to x : \overline{\tau_1}$ are not composable. Finally we write $A \asymp B$ if $A \odot B$ is defined.

C Affine/Linear Typing

In the following we give illustration of each rule in Figure 2. These rules are essentially the same as those in [5, 33, 34] except for mixing affinity and linearity, for which we need a careful attention for the input rules (where we recall that to ensure IO-alternation, we only allow free outputs appear in its body in all input rules [5, 33, 34]). We also assume $\uparrow_{\mathsf{L}} A$ in $(\mathsf{In}^{\downarrow_{\mathsf{L}}})$ is either singleton or empty ("unique answer per thread") [34].

(Zero) starts from empty type.
(Par) uses $\asymp$ and $\odot$ for controlling composition. For example, if P has type $x : ()^{\uparrow_{\mathsf{A}}}$ and Q has type $x : ()^{\uparrow_{\mathsf{L}}}$, then $P \mid Q$ is not typable because $()^{\uparrow_{\mathsf{A}}} \not\asymp ()^{\uparrow_{\mathsf{L}}}$.
(Res) allows hiding of a name only when its action mode is $*$ or $\mathbf{!}$ (which intuitively says channels of modes $\uparrow, \downarrow$ or $\mathbf{?}$ are always compensated by their duals before restricted).
(Weak) weakens $*$ and $\mathbf{?}$ since they allow the possibility of no action.
$(\mathsf{In}^{\downarrow_{\mathsf{A}}})$ does not record the causality because we do not care the termination at channel x. The side condition A^{-x} etc. is required to ensure affinity of x (see also § 2, Step 3).

($\mathsf{In}^{!_A}$) also does not record the causality. Note also that it cannot contain linear $\uparrow_L$/affine $\uparrow_A$ types in its body to ensure linearity and affinity; for example, if z is either linear or affine channels, then $!x(\vec{y}).(\overline{z} \mid Q)$ should be untypable because z is copied each interaction. Contrast to ($\mathsf{In}^{\downarrow_A}$), $?_L$-types can appear in its body since it allows the possibility of no action (cf. (Weak)).

($\mathsf{In}^{\downarrow_L}$) records the causality from linear input type $x : (\vec{\tau})^{\downarrow_L}$ to only linear output types (cf. **(G2)**).

($\mathsf{In}^{\uparrow_L}$) similarly records the causality from linear replicated input type to only $?_L$-types. The side condition A^{-x} is required to ensure no acyclicity.

(Out) does not suppress the body by prefix since output is asynchronous. Essentially the rule composes the output prefix and the body in parallel. This rule can be understood by translating $\overline{x}(\vec{y})P$ to $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle \mid P)$: suppose P has a type A. First we check $A \asymp x : (\vec{\tau})^{\overline{p}}$, then if defined, we hide $\vec{y}$ from $A \odot x : (\vec{\tau})^{\overline{p}}$, resulting $\overline{x}(\vec{y})P$ has type $A/\vec{y} \odot x : (\vec{\tau})^{\overline{p}}$.

D Transition Rules

To make the presentation simpler, we assume (w.l.o.g. up to $\equiv$) all prefixed terms to be primes, where P is a *prime with subject* x if either P is input-prefixed with subject x or $P \equiv \overline{x}(y_1..y_n)(P_1 \mid ... \mid P_n)$ such that each P_i is a prime with subject y_i. We also assume all l.h.s. processes are well-typed. We define the relation A *allows* l as the negation of: (1) $A(\mathsf{sbj}(l)) = \mathcal{M}_*$ or (2) l is output and $\mathsf{md}(A(\mathsf{sbj}(l))) \in \mathcal{M}_!$ [5]. $\mathsf{n}(l)$ is the set of names in l.

$$x(\vec{y}).P^{\,A} \xrightarrow{x(\vec{y})} P^{\,\vec{y}:\vec{\tau}\otimes A/x} \qquad (x : (\vec{\tau})^p \in A,\ p \in \mathcal{M}_\downarrow)$$

$$!x(\vec{y}).P^{A} \xrightarrow{x(\vec{y})} !x(\vec{y}).P|P^{\,\vec{y}:\vec{\tau}\otimes A} \qquad (x : (\vec{\tau})^p \in A,\ p \in \mathcal{M}_!)$$

$$\overline{x}(\vec{y})P^{\,A} \xrightarrow{\overline{x}(\vec{y})} P^{\,\vec{y}:\vec{\tau}\otimes A/x} \qquad (x : (\vec{\tau})^{\overline{p}} \in A)$$

$$\frac{P_1' \equiv_\alpha P_1 \quad P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad P_2 \equiv_\alpha P_2'}{P_1'^{\,A_1} \xrightarrow{l} P_2'^{\,A_2}} \qquad \frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad x \notin \mathsf{n}(l)}{(\boldsymbol{\nu}\,x)P_1^{A_1/x} \xrightarrow{l} (\boldsymbol{\nu}\,x)P_2^{A_2/x}}$$

$$\frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad A_1 \odot B \text{ allows } l}{P_1|Q^{\,A_1\odot B} \xrightarrow{l} P_2|Q^{\,A_2\odot B}} \qquad \frac{P_1^{A_1} \xrightarrow{l} P_2^{A_2} \quad Q_1^{B_1} \xrightarrow{\overline{l}} Q_2^{B_2}}{P_1|Q_1^{A_1\odot B_1} \xrightarrow{\tau} (\boldsymbol{\nu}\,\mathsf{bn}(l))(P_2|Q_2)^{\,A_2\odot B_2/\mathsf{bn}(l)}}$$

We omit rules for symmetric case of $|$. The rules for branching/selection and a reference agent are similarly defined [34, Appendix B].

E Extended Reduction

We define $\mapsto$ in the $\pi^{\mathsf{L}\oplus}$-calculus as the compatible relations on processes modulo $\equiv$ respectively generated by the following rules.

$$(\mathsf{Lin}) \quad C[\otimes_i \overline{x}(\vec{y})P_i]|x(\vec{y}).Q \mapsto C[\otimes_i(\boldsymbol{\nu}\,\vec{y})(P_i|Q)]$$

$$(\mathsf{Rep}) \quad C[\overline{x}(\vec{y})P]|!x(\vec{y}).Q \mapsto C[(\boldsymbol{\nu}\,\vec{y})(P|Q)] \mid !x(\vec{y}).Q$$

$$(\mathsf{Gar}) \quad (\boldsymbol{\nu}\,x)!x(\vec{y}).Q \mapsto \mathbf{0}$$

Here $C[\,]$ is an arbitrary context not binding x. Similarly for branching/selection [33, Appendix D.2]. Note that in (Lin), we contract linear redexes in all summands simultaneously. We also denote $\mapsto_{\mathsf{L}}$ for the extended reduction in π^{L}, which is defined deleting $\otimes_i$ from (Lin).