Deciding Deterministic Responsiveness and Closeness in π-calculus

Maxime Gamboni[*] António Ravara

September 21, 2006

Abstract

In order to write encodings of complex data structures in π-calculus one needs guarantees that certain fundamental properties of communication (together referred to as *discreetness*) are preserved when interacting with encoded processes.

These include *closeness* of the data transmission sub-process, *responsiveness* for data decoding requests, *uniqueness* of the transmitted value and *immutability* of a transmitted value.

We propose a bisimulation to relate processes indistinguishable by such a discreet environment.

Discreetness itself being undecidable we define a type system of π-calculus processes with stronger but decidable constraints, and prove that typable processes are discreet.

1 Introduction

This work originated from a need to model complex data exchange in a way that "looks" atomic.

More specifically, we want to be able to encode in the polyadic π-calculus [MPW92], [SW01], [Mil93] (that only allows transmitting names) expressions such as $\overline{a}\langle\xi\rangle$ where ξ is an arbitrarily complex structure, preserving fundamental properties of data transmission.

To make our requirements precise, we need to further specify the data exchange protocol.

Encodings can be split into three broad classes:

[*]This work was partially supported by FEDER, the EU IST proactive initiative FET-Global Computing (project Sensoria, IST-2005-16004), and Fundação para a Ciência e a Tecnologia (via CLC and the project Space-Time-Types, POSC/EIA/55582/2004).

a. The sender keeps the data and the receiver requests it every time it needs it

b. The data is lazily reconstructed on the receiver side and transmitted according to needs

c. The data is first fully transmitted and reconstructed on the receiver side before proceeding

Method a requires taking special care to ensure data consistency accross requests while method b and c require the encoding to be more complex (because it requires extra machinery for mirroring the remote data). Moreover, method c only works if the data is finite, which may or may not be the case (the source calculus might have a concept of recursive data for instance), so extra checks are needed to validate environments. On the other hand, methods a and b require the type system to make sure the sender can't infer if and when the data is accessed by the receiver.

As a criteria for choosing between the three above approaches we decided that a simple encoding is preferable, even at the price of making the type system more complex (as the same type system may be used for other encodings and source calculi), so we choose method a.

Consider the following prototypical process:
$(\overline{a}\langle u\rangle \,|\, [\![\, \xi \,]\!]_u) \mid (a(x).\overline{x}\langle r\rangle \,|\, r(v).P)$, where a piece of data accessible through the server at u is sent over a. The receiver then sends a decoding request and receives the reply v at r. The name u has what we call an *ω-class* and needs to satisfy the following requirements:

1. Once sent, the transmitted data is fully determined.

2. the receiver can access it arbitrarily often and always gets the same result.

3. The sender can't know when and how many times the data is accessed.

4. A decoding request is always replied to after a finite time.

Of course these are about the data that is actually part of ξ — if it contains references or pointers then only the pointer itself has to satisfy the above, not the pointed structure. The requirements on r, which is said of *linear* class, are different: As it is specific to a particular request, its input does not need to be replicated. However, exactly one output must happen (as we want one reply to a request). For simplicity we require exactly one input and one output to be available at some point on r. We need to permit side effects and non-deterministic behaviour outside of u's input side, even if it is prefixed by r. Indeed, a process having received the reply to a decoding request should not suffer additional constraints. Finally a, not having any particular requirement, is said *plain*.

These three name classes and the associated requirements are enough to express a wide range of encodings. For instance, x being linear and t, f plain, a boolean value can be encoded at an ω-name a as: $\mathrm{True}(a) = \mathbf{!}\, a(x,t,f).\overline{x}\langle t\rangle$ and $\mathrm{False}(a) = \mathbf{!}\, a(x,t,f).\overline{x}\langle f\rangle$. ($\texttt{if}\ a\ \texttt{then}\ T\ \texttt{else}\ F$) is then done using $(\boldsymbol{\nu} t,f)\,\overline{a}\langle x,t,f\rangle.(x(p).\bar{p} \mid t.T \mid f.F)$. Note that the semantics of ω and linear classes do not force the reply to be either t or f, as $\mathbf{!}\, a(x,t,f).\overline{x}\langle p\rangle$ (p plain) is considered valid. *Choice types*, which we leave as future work, could enforce it.

Our setting is both a specialisation (in that we do not use affine modes) and a generalisation of [YBH04].

The novelty lies in the introduction of *plain* modes that can freely interact with ω or linear modes, yet having almost no constraints. To strong normalisation (responsiveness in our terminology) we add closeness (lack of side effects) and determinism.

Here is a brief overview of this paper.

In Section 2 we rigorously specify the above requirements (which we group under the term of *discreetness*), and, in parallel, define *discreet bisimulations* [Mil89], which relate processes that can not be distinguished by discreet environments. Clearly, discreetness is not decidable, which is why, in Section 3, we propose a correct (but necessarily incomplete) type system that only types processes complying to these properties. Section 4, finally, contains the proofs of various properties and theorems mentioned in the previous chapters.

As a future development, the theory and type system introduced in this paper can be extended with *choice types* that can type sums. A concrete application of this work would be to prove the equivalence of TyCO and π_{a}^{V} [GNR04].

2 Semantic Requirements

In this section we give a rigorous definition of our semantic requirements.

2.1 Syntax

We are working in the synchronous polyadic π-calculus with full replication and mixed sums.

A tilde placed over a symbol stands for a sequence, indexed from 1 to some finite number n. The same symbol without the tilde and with an index represents an individual element of the sequence. For instance $\tilde{x} = x_1, x_2, \ldots, x_n$ for some $n \geq 0$.

A process P is one of $\mathbf{0}$, $P|Q$, $(\boldsymbol{\nu} x)\, P$, $a(\tilde{x}).P$, $\overline{a}\langle\tilde{x}\rangle.P$ and $\mathbf{!}\, P$. One syntactical requirement is that in input processes $a(\tilde{x}).P$, a may not be part of $\tilde{x}$, and all names in $\tilde{x}$ must be distinct. Processes such as $\overline{x}\langle x, x\rangle$ are allowed, however.

The calculus only has a free output but bound outputs can be simulated with $(\boldsymbol{\nu}\tilde{y})\, \overline{a}\langle\tilde{x}\rangle.P_{\tilde{y}}$ where $\tilde{y} \subseteq \tilde{x}$ and $P_{\tilde{y}}$ a process listening at all names in $\tilde{y}$.

The set of free names in a process is defined as usual:

P	$\mathrm{fn}(P)$
$\mathbf{0}$	$\emptyset$
$P \mid Q$	$\mathrm{fn}(P) \cup \mathrm{fn}(Q)$
$(\boldsymbol{\nu} x)\, P$	$\mathrm{fn}(P) \setminus \{x\}$
$P + Q$	$\mathrm{fn}(P) \cup \mathrm{fn}(Q)$
$a(\tilde{x}).P$	$\{a\} \cup (\mathrm{fn}(P) \setminus \tilde{x})$
$\overline{a}\langle\tilde{x}\rangle.P$	$\{a\} \cup \tilde{x} \cup \mathrm{fn}(P)$
$\mathbf{!}\, P$	$\mathrm{fn}(P)$

Capture avoiding substitution $P\{^{\tilde{x}}/_{\tilde{y}}\}$ is standard.

2.2 Channel Types

The definition of discreetness depends on the *types* of involved names. They are defined among the lines of [YBH04].

An Action Mode describes the kind of interactions a process does at a name.

Definition 2.2.1 (Action Modes) *An* action mode *m is an element of the set $\{\downarrow_{\mathtt{1}}, \uparrow_{\mathtt{1}}, \bigstar_{\mathtt{1}}, \downarrow_{\omega 0}, \downarrow_\omega, \uparrow_\omega, \bigstar_\omega, \mathtt{p}\}$.*

Semantic *modes are $\{\downarrow_{\mathtt{1}}, \uparrow_{\mathtt{1}}, \bigstar_{\mathtt{1}}, \downarrow_\omega, \uparrow_\omega, \mathtt{p}\}$,* input *and* output *modes are respectively $m^{\downarrow} = \{\downarrow_{\mathtt{1}}, \downarrow_\omega, \mathtt{p}\}$ and $m^{\uparrow} = \{\uparrow_{\mathtt{1}}, \uparrow_\omega, \mathtt{p}\}$,* parameter *modes are $m^{\downarrow} \cup m^{\uparrow}$,* inert *modes are $m^{\bigstar} = \{\bigstar_{\mathtt{1}}, \bigstar_\omega, \uparrow_\omega, \mathtt{p}\}$,* complete *modes are $m^{\mathrm{k}} = \{\bigstar_{\mathtt{1}}, \downarrow_\omega, \mathtt{p}\}$, and* resource *modes are $m^{\mathrm{r}} = \{\downarrow_{\mathtt{1}}, \uparrow_{\mathtt{1}}, \bigstar_{\mathtt{1}}, \downarrow_\omega, \downarrow_{\omega 0}, \bigstar_\omega\}$. The* complement *$\overline{m}$ of m is obtained by swapping arrows. $\overline{\mathtt{p}} = \mathtt{p}$ and $\overline{\downarrow_{\omega 0}} = \uparrow_\omega$.*

Modes $\downarrow_{\mathtt{1}}$, $\uparrow_{\mathtt{1}}$, $\bigstar_{\mathtt{1}}$ respectively stand for input, output and uncomposable, on a linear name; $\downarrow_\omega$, $\downarrow_{\omega 0}$, $\uparrow_\omega$, $\bigstar_\omega$ on an ω-name, input, unreplicated input, output and uncomposable; $\mathtt{p}$ is for any kind of interaction over a plain name.

The $\bigstar_{\mathtt{1}}$ mode is used when a process does both input and output at a linear name (uncomposable means that it does not compose with a process where the name is free); $\downarrow_{\omega 0}$ is used as a "temporary" mode and means a non-replicated ω-input, that is completed once the process gets replicated; $\bigstar_\omega$ is used in cases such as $P|\overline{p}\langle u\rangle.!\,u$, where p is plain and u is ω. u, having mode $\bigstar_\omega$, may be used as a parameter by the corresponding receivers at p, but may not otherwise be referenced in P. Note that $\downarrow_\omega$ and $\uparrow_\omega$ correspond to ! and ? in [YBH04], respectively.

The *class* of a mode tells whether it is plain, linear or ω.

Parameter modes are those that an input can provide at its parameters. Inert modes are actions that can be provided by weakening. A complete mode is one that can be bound, and resource modes are those that can be depended upon.

Definition 2.2.2 (Channel Types) *Channel types are generated by this grammar:* $\sigma ::= (\sigma_1 \cdots \sigma_n)^m$*, where* σ_i *are the parameter types,* $n \geq 0$ *and* m *is an action mode. We write* $\mathbf{md}(\sigma)$ *to mean the action mode of a type.*

This function is naturally generalised to channel types sets, as $\mathbf{md}(\tilde{\sigma}) \stackrel{\text{def}}{=} \{m : \exists \sigma_i \in \tilde{\sigma} : \mathbf{md}(\sigma_i) = m\}$.

All action modes in a type except the outermost have to be parameter action modes. The complement $\bar{\sigma}$ *of a channel type* σ *is obtained by replacing the outermost mode by its complement.*

ϵ *stands for the neutral type . Brackets are omitted when there are no parameters.*

The mode of parameters are as seen from the receiving end. (unlike [YBH04] — The reason why we write the parameter polarity independently from the end is that plain names are managed without polarity.)

For instance $\sigma = (\downarrow_1)^{\downarrow_1}$ would be a type for a in $a(x).\bar{x}$. The complement $\bar{\sigma} = (\downarrow_1)^{\uparrow_1}$ represents the corresponding output $\overline{a}\langle x\rangle.x$.

Definition 2.2.3 (Channel Composition) *The channel composition operator* $\odot$ *is the* partial *commutative operator on channel types derived from the following rules.*

$\sigma \odot \sigma = \sigma$ *if* $\mathbf{md}(\sigma) \in \{\uparrow_\omega, \mathtt{p}\}$
$\sigma \odot \overline{\sigma} = \sigma$ *if* $\mathbf{md}(\overline{\sigma}) \in \{\uparrow_\omega, \mathtt{p}\}$
$(\tilde{\sigma})^{\uparrow_1} \odot (\tilde{\sigma})^{\downarrow_1} = (\tilde{\sigma})^{\bigstar_1}$
$\sigma \odot \epsilon = \sigma$.

The reverse operator, channel subtraction, is used when a resource is consumed (or migrates out of a process). We use a conservative definition, where non-linear modes never disappear.

Definition 2.2.4 (Channel Subtraction) *Channel subtraction is defined only between identical types. Then,* $(\tilde{\sigma})^{m_1} \setminus (\tilde{\sigma})^{m_2}$ *is equal to* $(\tilde{\sigma})^{m}$*, where* m *is as follows.*

1. $m_1 = m_2 \in \{\uparrow_\omega, \downarrow_\omega, \mathsf{p}\}$ *implies* $m = m_1$.

2. $m_1 = m_2 \in \{\downarrow_1, \uparrow_1\}$ *implies* $m = \epsilon$.

The subtraction is otherwise undefined.

Note that we do not define subtraction in cases such as $\bigstar_1 \setminus \downarrow_1$ or $\downarrow_\omega \setminus \uparrow_\omega$, because they represent transitions that are not observable from the outside.

These operators are conventionally evaluated from left to right, e.g. $\sigma_1 \odot \sigma_2 \setminus \sigma_3$ is read as $(\sigma_1 \odot \sigma_2) \setminus \sigma_3$ and $\sigma_1 \setminus \sigma_2 \odot \sigma_3$ as $(\sigma_1 \setminus \sigma_2) \odot \sigma_3$.

Also note that in case $\sigma \setminus \sigma'$ is defined, we have $\sigma \setminus \sigma' \odot \sigma' = \sigma$. However the opposite rule $\sigma \odot \sigma' \setminus \sigma' = \sigma$ usually does not hold.

Σ being a mapping of names to channel types, these are generalised as follows:

$$(\Sigma \odot a : \sigma)(x) = \begin{cases} \Sigma(x) \odot \sigma & \text{if } x = a \\ \Sigma(x) & \text{if } x \neq a \end{cases} \qquad (\Sigma \setminus a : \sigma)(x) = \begin{cases} \Sigma(x) \setminus \sigma & \text{if } x = a \\ \Sigma(x) & \text{if } x \neq a \end{cases}$$

2.3 Operational Semantics

Table 1 defines the operational semantic of our target calculus. (COM$_2$), (PAR$_2$) and (SUM$_2$) are omitted, and are the symmetrical versions of the corresponding 1-indexed rules.

$P \equiv_\alpha Q$ means Q can be obtained from P using α-renaming.

The set of valid labels are τ for silent operation (also known as reduction), $a(\tilde{x})$ for input, and $(\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle$, where $\tilde{z} \subseteq \tilde{y} \setminus a$, for output. When $\tilde{z} = \emptyset$ we omit the restriction operator.

The free output objects of a transition label are: $\mathsf{fo}(\tau) = \mathsf{fo}(a(\tilde{x})) = \emptyset$ and $\mathsf{fo}((\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle) = \tilde{y} \setminus \tilde{z}$.

The bound output objects of a transition label are: $\mathsf{bo}(\tau) = \mathsf{bo}(a(\tilde{x})) = \emptyset$ and $\mathsf{bo}((\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle) = \tilde{z}$

The bound (either input or output) names of a transition label are : $\mathrm{bn}(\tau) = \emptyset$, $\mathrm{bn}(a(\tilde{x})) = \tilde{x}$ and $\mathrm{bn}((\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle) = \tilde{z}$.

Finally, the set of names of a label is: $\mathrm{n}(\tau) = \emptyset$, $\mathrm{n}(a(\tilde{x})) = \{a\} \cup \tilde{x}$ and $\mathrm{n}((\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle) = \{a\} \cup \tilde{y}$.

Two transitions labels μ and μ' are said *equivalent*, written $\mu \equiv \mu'$ if they can be obtained from each other by changing bound names. For instance

$$\frac{}{a(\tilde{x}).P \xrightarrow{a(\tilde{y})} P\{^{\tilde{y}}/_{\tilde{x}}\}}\ (\textsc{Inp}) \qquad \frac{}{\overline{a}\langle\tilde{y}\rangle.P \xrightarrow{\overline{a}\langle\tilde{y}\rangle} P}\ (\textsc{Out})$$

$$\frac{P \xrightarrow{\mu} P' \quad x \in \mathsf{fo}(\mu)}{(\boldsymbol{\nu}x)\,P \xrightarrow{(\boldsymbol{\nu}x)\,\mu} P'}\ (\textsc{Open}) \qquad \frac{P \xrightarrow{\mu} P'}{!\,P \xrightarrow{\mu} !\,P|P'}\ (\textsc{Rep})$$

$$\frac{P \xrightarrow{a(\tilde{y})} P' \qquad Q \xrightarrow{(\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{y}\rangle} Q' \qquad \tilde{z}\cap \mathrm{fn}(P)=\emptyset}{P|Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z})\,(P'|Q')}\ (\textsc{Com}_1)$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathsf{bo}(\mu)\cap\mathrm{fn}(Q)=\emptyset}{P|Q \xrightarrow{\mu} P'|Q}\ (\textsc{Par}_1) \qquad \frac{P\xrightarrow{\mu}P'}{P+Q\xrightarrow{\mu}P'}\ (\textsc{Sum}_1)$$

$$\frac{P \xrightarrow{\mu} P' \quad x\notin \mathrm{n}(\mu)}{(\boldsymbol{\nu}x)\,P \xrightarrow{\mu} (\boldsymbol{\nu}x)\,P'}\ (\textsc{Res}) \qquad \frac{P\equiv_\alpha Q \qquad Q\xrightarrow{\mu}Q'}{P\xrightarrow{\mu}Q'}\ (\textsc{Alpha})$$

Table 1: Labelled Transition System

$a(x,y) \equiv a(z,t)$ and $(\boldsymbol{\nu}x)\,\overline{a}\langle x,y\rangle \equiv (\boldsymbol{\nu}z)\,\overline{a}\langle z,y\rangle \not\equiv (\boldsymbol{\nu}z)\,\overline{a}\langle z,w\rangle$. On input labels, this renaming occurs independently of the subject, e.g. $a(x,y) \equiv a(a,y)$.

We write $P \longrightarrow Q$ to mean $P \xrightarrow{\tau} Q$. $\Rightarrow$ is the transitive reflexive closure of relation $\longrightarrow$, and $P \overset{\mu}{\Longrightarrow} Q$ means $P \Rightarrow\xrightarrow{\mu}\Rightarrow Q$. $P \overset{\hat{\mu}}{\Longrightarrow} Q$ means $P \Rightarrow Q$ if $\mu = \tau$, $P \overset{\mu}{\Longrightarrow} Q$ otherwise.

The discreetness notions defined later require some knowledge about the channel types of involved names, so from now on processes will always be associated with a type mapping.

We lift this transition system to *typed processes* $(\Sigma; P)$ where Σ is a mapping of names to channel types and P a process. Channel type operators $\odot$ and $\setminus$ are trivially generalised to such mappings, treating a missing name on either side as being mapped to the neutral type ϵ.

Let Σ and Σ' be name-channel type mappings, P and P' processes and μ a transition label such that $P \xrightarrow{\mu} P'$. When $\mu \neq \tau$, we assume $\Sigma(a) = (\tilde{\sigma})^m$, and $\tilde{\sigma}_z$ is the subset of $\tilde{\sigma}$ corresponding to $\tilde{z}$.

If $\mu = \tau$: $(\Sigma; P) \xrightarrow{\mu} (\Sigma; P')$

If $\mu = a(\tilde{x})$: $(\Sigma; P) \xrightarrow{\mu} ((\Sigma \ \setminus\ a : (\tilde{\sigma})^{m^{\downarrow}}) \ \odot\ \tilde{x} : \overline{\tilde{\sigma}}; P')$

If $\mu = (\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{x}\rangle$: $(\Sigma; P) \xrightarrow{\mu} ((\Sigma \ \setminus\ a : (\tilde{\sigma})^{m^{\uparrow}}) \ \odot\ \tilde{z} : \tilde{\sigma}_z; P')$

If any of these operators is undefined then so is the transition itself. Note that $\Sigma \odot \tilde{x} : \tilde{\sigma}$ actually means $\Sigma \odot x_1 : \sigma_1 \odot \cdots \odot x_n : \sigma_n$, which is then well defined in case names in $\tilde{x}$ are not all distinct. Typed process may have less transitions than the process itself:

First, not all process transitions are observable from outside (if it is one end of a linear communication, or an ω-request whose receiver is also in the process);

Second case is when the transition is not safe and must not be tested. (This is the case for instance in $a|b(x).x \xrightarrow{b(a)} a|a$.)

In the following definition, $(\boldsymbol{\nu}\tilde{x})\,\Sigma$ is defined if $\forall x_i \in \tilde{x}$, $\mathbf{md}(\Sigma(x_i)) \in \{\bigstar_1, \downarrow_\omega, \mathtt{p}\}$, and is then equal to $\Sigma\mid_{\mathrm{dom}(\Sigma)\setminus\tilde{x}}$. The $\cdot$ operator used below is based on the channel type operator defined as $\sigma_1 \cdot \sigma_2 = \sigma_1$ if $\sigma_1 = \sigma_2$, and $\sigma_1 \odot \sigma_2$ otherwise.

Definition 2.3.1 (Type Consistency) *The set $\mathbb{S}$ of $(\Sigma; P)$ pairs (Σ being type mappings and P processes) is the smallest one generated from the following rules.*

$(\emptyset; \mathbf{0}) \in \mathbb{S}$

If $(\Sigma_i; P_i) \in \mathbb{S}$ for both $i = 1, 2$, then

$(\Sigma_1 \cdot \Sigma_2; P_1|P_2) \in \mathbb{S}$,

$((\boldsymbol{\nu}a)\,\Sigma_1; (\boldsymbol{\nu}a)\,P_1) \in \mathbb{S}$,

$(\Sigma_1; !\,P_1) \in \mathbb{S}$.

Having $\sigma = (\tilde{\sigma})^m$: $(a : \sigma^\uparrow \cdot \tilde{x} : \overline{\tilde{\sigma}} \cdot \Sigma_1; \overline{a}\langle\tilde{x}\rangle.P_1) \in \mathbb{S}$, and

$((\boldsymbol{\nu}\tilde{x})\,\big(a : \sigma^\downarrow \cdot \tilde{x} : \tilde{\sigma} \cdot \Sigma_1\big)\,; a(\tilde{x}).P_1) \in \mathbb{S}$.

$\forall \Sigma'$ *s.t.* $\forall x \in \mathrm{n}(\Sigma') : \mathbf{md}(\Sigma'(x)) \in m^\bigstar$: $(\Sigma_1 \cdot \Sigma'; P_1) \in \mathbb{S}$

We say that Σ is consistent *for P when $(\Sigma; P) \in \mathbb{S}$.*

The definition for $\mathbb{S}$ is used to make sure the processes contain every non-inert mode mentioned in the type, to prevent typings such as $(l : \bigstar_1; \bar{l})$, in which l's input mode is not present in the process. Allowing any inert mode is required for safety, for instance with processes such as $l|\bar{l} \rightarrow \mathbf{0}|\mathbf{0}$, where $l : \bigstar_1$ even after the transition.

Note that consistency is not preserved under τ-transitions. For instance $(u :\downarrow_\omega; u.u) \in \mathbb{S}$ but after two $\xrightarrow{u}$ transitions reduces to $(u :\downarrow_\omega; \mathbf{0})$ which is not in $\mathbb{S}$.

Any process may be typed in different ways. The differences arise from the weakening rule (any inert type may be added to any name), from names that are not used as channels (so neither the parameter types nor parameter count can be inferred by examining the process) and by changing name classes (for instance in $\bar{x}$, x could be any of $\uparrow_1$, $\uparrow_\omega$ or $\mathtt{p}$). It may be expressed as follows:

The channel type definition is generalised to *incomplete channel types* where an incomplete channel type σ is either a symbol τ_i, a complement symbol $\overline{\tau_i}$ or is of the form $(\tilde{\sigma})^m$ where each σ_i is an incomplete channel type. Such an incomplete channel type can be *instantiated* with $\{{}^{\tilde{\sigma}}/_{\tilde{\tau}}\}$ mapping all τ_i symbols to regular channel types, simply replacing each τ_i by the corresponding σ_i and $\overline{\tau_i}$ by $\overline{\sigma_i}$. Channel type mappings Σ are generalised the same way.

We write $T \succcurlyeq \Sigma$ if there is Σ' containing only inert modes, as well as a mapping $\tilde{\tau} \mapsto \tilde{\sigma}$ such that $\Sigma = T\{{}^{\tilde{\sigma}}/_{\tilde{\tau}}\} \cdot \Sigma'$.

Σ being a type mapping and T an incomplete type mapping, we say that *Σ's name classes matches those of T* if there is no name shared by Σ and T that is mapped to different classes (plain, ω or linear) by the two mappings.

Lemma 2.3.2 (Relating Consistent Types) *Let $(\Sigma; P) \in \mathbb{S}$.*

Then there is an incomplete type mapping T whose name classes match those of Σ and s.t. $(\Sigma'; P) \in \mathbb{S}$ where Σ''s name classes match those of T if and only if $T \succcurlyeq \Sigma'$.

2.4 Template processes

To test processes that make requests to external ω-names we compose them with generic contexts providing *template* inputs for the requested names. The definition is as follows.

Definition 2.4.1 (Template Process) *$\sigma = (\sigma_1 \cdots \sigma_n)^m$ being a channel type, $\mathrm{L}_\sigma(a)$ is a process that provides an input or output at a, matching σ.*

If $m \in m^\bigstar$, $\mathrm{L}_\sigma(a) = \mathbf{0}$.

If $m = \uparrow_1$, $\mathrm{L}_\sigma(a) = (\boldsymbol{\nu}\tilde{y})\,\overline{a}\langle\tilde{a}\rangle.\prod_i \mathrm{L}_{\sigma_i}(a_i)$, $\tilde{y}$ being the subset of $\tilde{a}$ with mode `p`.

If $m{=}\downarrow_1$, $\mathrm{L}_\sigma(a) = a(\tilde{a}).\prod_i \mathrm{L}_{\overline{\sigma_i}}(a_i)$, and

if $m{=}\downarrow_\omega$, $\mathrm{L}_\sigma(a) = \boldsymbol{!}\,a(\tilde{a}).\prod_i \mathrm{L}_{\overline{\sigma_i}}(a_i)$.

Two things are to be noted in the above definition.

First, being a recursive definition, it may define an infinitely nested process. Even in the case of recursively defined types, it would not be possible to use a recursive process and keep full generality. Consider a dialogue between two processes, where exchanged names are only of two different types. Even though there is a finite number of message types, the number of message values is not bounded, and may not be modelled by a finite process.

Lemma 2.4.2 (Template Typing) *σ being a channel type, we define Σ so that for each plain type σ_0 contained in σ at position $\tilde{i}$ and not itself contained in another plain type, $\Sigma(a_{\tilde{i}}) = \sigma_0$. Then $\mathrm{T}_\sigma(a) \overset{\text{def}}{=} a : \sigma \odot \Sigma$ is an consistent type for $\mathrm{L}_\sigma(a)$).*

2.5 Observability

ω-names can be considered as a form of *storage* of *immutable* data in the process. So we are going to define a generic representation of data stored in a process, which allows to easily express our requirements.

The following is the grammar we are going to use to write the values of this function. $\xi \ ::= \ p \ \ | \ \ \langle\tilde{\xi}\rangle$, where p is a name.

We define observability in two steps. We will first assume the process does not depend on external resources to provide observable data, and then generalise it for any process (like forwarders $!\, u(x).\overline{v}\langle x\rangle$, branches: $!\, u(x,y).(\overline{v}\langle x\rangle \mid \overline{v}\langle y\rangle)$ or simple dependencies: $\bar{l}.!\, a \mid !\, b(x).\bar{a}.\bar{x}$).

Definition 2.5.1 (Observability) *Let $(\Sigma; P)$ be a typed process and a a name with $\Sigma(a) = ((\tilde{\sigma}_1)^{m_1}, \cdots, (\tilde{\sigma}_n)^{m_n})^m$. The data* directly observable *at a in P is ξ defined as follows.*

If $m \in m^\bigstar$ then $\xi = \langle\rangle$.

Otherwise, assume $P \overset{\mu}{\Longrightarrow} P'$ where μ is either $a(\tilde{x})$ or $\overline{a}\langle\tilde{x}\rangle$ depending on m's polarity. In case of input we set $\forall i : x_i = a_i$. Let $P''_i = P' \mid \prod_{j \neq i} \mathrm{L}_{\sigma_j}(a_j)$ where $\sigma_j = (\tilde{\sigma}_j)^{m_j}$ if μ is an input, $\overline{(\tilde{\sigma}_j)^{m_j}}$ otherwise. Then $\xi = \langle\xi_1 \cdots \xi_n\rangle$ where ξ_i $(0 \leq i \leq n)$ is defined as follows.

if $m \in m^\uparrow$ and $m_i = \mathtt{p}$ then $\xi_i = a_i$,

if $m \in m^\uparrow$ and $m_i = \uparrow_\omega$ then $\xi_i = \langle\rangle$,

if $m \in m^\uparrow$ and $m_i \notin m^\bigstar$ then ξ_i is the data directly observable at a_i in P''_i.

if $m \in m^\downarrow$ and $\overline{m_i} \in m^\bigstar$ then $\xi_i = \langle\rangle$,

if $m \in m^\downarrow$ and $\overline{m_i} \notin m^\bigstar$ then ξ_i is the data directly observable at a_i in P''_i.

If there is no such μ transition then ξ is undefined. On the other way round if there are more than one such transition then ξ can have different values.

Let $\tilde{r} = \{r : \overline{\mathbf{md}(\Sigma(r))} \notin m^\bigstar\}$. Then $\Omega^\Sigma_P(a)$ is the data directly observable at a in process $P \mid \prod_{r \in \tilde{r}} \mathrm{L}_{\overline{\Sigma(r)}}(r)$.

Note the use of indexed a as parameters. This is needed for processes that return received names, like $True(x)$ and $False(x)$ (p. 3) for which the data directly observable at x are $\langle\langle x_2\rangle, \langle\rangle, \langle\rangle\rangle$ and $\langle\langle x_3\rangle, \langle\rangle, \langle\rangle\rangle$. If a is itself already indexed (say $a = x_1$), then indexed forms of a are written with a comma-separated sequence of indexes, for instance as in $a_2 = x_{1,2}$. When computing ξ_i we compose the process with template processes for all other parameters, to handle processes such as $!\, u(x, y).\overline{y}\langle x\rangle$, that query part of the request when composing the reply. Similarly, external dependencies are provided using template processes. Requests to external ω-names are handled uniformly, as the replies are entirely determined by the channel name (in $P = a(x, y).\overline{b}\langle x\rangle|\overline{b}\langle y\rangle$, assuming b has mode $\uparrow_\omega$, $\Omega^{\Sigma}_{P}(a)$ contains the same name twice: $\langle\langle b_{1,1}\rangle, \langle b_{1,1}\rangle\rangle$).

Observability behaves according to our semantic definitions of name classes:

- If a process may not depend on an external plain name to reply to requests, due to the unreliable nature of plain modes. This is reflected by the plain name template being equal to $\mathbf{0}$, i.e. ignoring all these requests for obtaining observable data.
- Requests to external linear names are however accepted, as long as a given name is used at most once. This is reflected in the template process by having non-replicated inputs and outputs for linear names. If the tested process attempts more than one request on the same linear name then the second request will go unanswered, which, like in the case of plain names above, makes the observable data undefined.
- Finally, requests to external ω-names are handled in a deterministic and uniform way by template processes, as the replies are entirely determined by the channel's name and type. This is visible for instance in process $a(x, y).\overline{b}\langle x\rangle|\overline{b}\langle y\rangle$, for which (assuming b has mode $\uparrow_\omega$) observable data at both a's parameters is the same ($\langle\langle b_{1,1}\rangle, \langle b_{1,1}\rangle\rangle$).

2.5.1 Examples

Let $\Sigma(a) = \sigma = \left(\mathtt{p},\ (\mathtt{p})^{\uparrow_1},\ (\mathtt{p})^{\downarrow_1},\ \left((\mathtt{p})^{\downarrow_1}\right)^{\uparrow_1}\right)^{\downarrow_\omega}$.

Let P be a process providing an input at a and such that Σ is consistent for P. Then $\Omega^{\Sigma}_{P}(a)$ will be of the form $\langle\langle\rangle, \langle\langle\rangle\rangle, \langle x\rangle, \langle\langle y\rangle\rangle\rangle$.

For instance if $P = !\, a(p, b, c, d).\left(b(q) \mid \overline{c}\langle r\rangle \mid d(l).\overline{l}\langle s\rangle\right)$ then a has an observable value in P, which is $\Omega^{\Sigma}_{P}(a) = \langle\langle\rangle, \langle\langle\rangle\rangle, \langle r\rangle, \langle\langle s\rangle\rangle\rangle$.

We consider that an output on an ω-name carries no information because it is supposed to have no side effects on the input side. For example a value associated with a channel of type $\sigma = \left((\mathtt{p})^{\downarrow_\omega}\right)^{\uparrow_1}$ will always be $\langle\langle\rangle\rangle$, and is for instance observable at a in $a(u).\mathbf{0}$ or in $a(u).(\overline{u}\langle p\rangle|\overline{u}\langle q\rangle)$.

If $P = \mathrm{L}_\sigma(a)$, then $\Omega^\Sigma_P(a)$ is of the form required by the type and in contains only plain names like $a_{i,\ldots,j,k}$ where the indexes specify the position in the record. In this case:

$$P = \mathbf{!}\, a(a_1, a_2, a_3, a_4).\,(a_2(a_2, 1) \mid \overline{a_3}\langle a_{3,1}\rangle \mid a_4(a_{4,1}).\overline{a_{4,1}}\langle a_{4,1,1}\rangle)$$

and

$$\Omega^\Sigma_P(a) = \langle\langle\rangle, \langle\langle\rangle\rangle, \langle a_{3,1}\rangle, \langle\langle a_{4,1,1}\rangle\rangle\rangle$$

Observable data may come from three places: decided by the process itself (x in the example below), delegated to a remote name (y) or bounced from the parameters (z). The following example illustrates this.

Let $\sigma_q = \left((\mathtt{p})^{\downarrow_1}\right)^{\downarrow_1}$ $\sigma' = \left((\mathtt{p})^{\downarrow_1}, (\mathtt{p})^{\downarrow_1}, (\mathtt{p})^{\downarrow_1}, \sigma_q\right)^{\downarrow_1}$.

Now let $Q = b(x, y, z, t).\overline{x}\langle p\rangle \mid \overline{c}\langle y\rangle \mid \overline{t}\langle z\rangle$, in which $b : \sigma'$ and $c : \sigma_q^{\uparrow}$.

We then have $\Omega^\Sigma_Q(b) = \langle\langle p\rangle, \langle c_{1,1}\rangle, \langle b_{4,1,1}\rangle, \langle\langle\rangle\rangle\rangle$.

2.6 Bisimilarity

We are now going to define a bisimilarity relation on (typed) processes. There are two applications for this.

One, which is the main contribution of this paper, is that a bisimilarity relation allows validating calculus encodings using a full abstraction property. Indeed, the usual weak bisimilarity relation would be too strong because it allows observing the details of data decoding protocols.

Additionally, such a relation allows to define discreetness easily: We express the absence of side effects caused by a request on a reserved name as bisimilarity of the processes before and after the request.

We restrict ourselves to $\mathbb{S}$ (see Definition 2.3.1) because inconsistent types may prevent some transitions to be tested, thereby wrongly seeing two processes as bisimilar. For instance $P_0 = \overline{a}\langle\tilde{x}\rangle.P$, where P is any process with $x \notin \mathrm{fn}(P)$.

When this process is associated with the types $a : \left(()^{\uparrow_\omega}\right)^{\mathtt{P}}$ and $x : ()^{\downarrow_\omega}$, the transition $P_0 \xrightarrow{\overline{a}\langle\tilde{x}\rangle} P$ would not be tested, and as we will see in the definition for discreet processes, any process of this form would then be considered discreet.

Definition 2.6.1 (Discreet Bisimulation) *A symmetric relation $\mathcal{R}$ on $\mathbb{S}$ is a* discreet bisimulation *if* $(\Sigma_P; P)\mathcal{R}(\Sigma_Q; Q)$ *implies the following.*

Below, p, l and u respectively have a plain, linear and ω-mode in the corresponding type mapping. $a \in \mathrm{dom}(\Sigma_P) \cap \mathrm{dom}(\Sigma_Q)$ implies $\Sigma_P(a)$ and $\Sigma_Q(a)$ have the same class.

1. *If* $(\Sigma_P; P) \xrightarrow{\mu} (\Sigma'_P; P')$ *and* $\mu \in \{\tau, p(\tilde{x}), (\boldsymbol{\nu}\tilde{z})\,\overline{p}\langle\tilde{x}\rangle, l(\tilde{x}), (\boldsymbol{\nu}\tilde{z})\,\overline{l}\langle\tilde{x}\rangle\}$ *then* $\exists \mu' \equiv \mu$ *s.t.* $(\Sigma_P; P) \xrightarrow{\mu'} (\Sigma''_P; P'')$, $(\Sigma_Q; Q) \overset{\hat{\mu}'}{\Longrightarrow} (\Sigma'_Q; Q')$ *and* $(\Sigma''_P; P'')\mathcal{R}(\Sigma'_Q; Q')$.

2. $(P; \Sigma_P) \xrightarrow{u(\tilde{x})} (\Sigma'_P; P')$ *implies:*
   - $(\Sigma'_P; P')\mathcal{R}(\Sigma'_P; P')$,
   - $\exists! \xi$ *s.t.* $\Omega^{\Sigma}_P(u) = \xi$, *and*
   - $(\Sigma_P; P)\mathcal{R}((\boldsymbol{\nu}\tilde{x})\,\Sigma'_P; (\boldsymbol{\nu}\tilde{x})\,P')$,
   - $(\Omega^{\Sigma}_P(u) = \xi) \Rightarrow (\Omega^{\Sigma}_Q(u) = \xi)$.

3. $\forall u : \sigma = \Sigma_P(u)$ *and* $\mathbf{md}(\sigma) = \uparrow_\omega$ *implies:*
   $((\boldsymbol{\nu}u)\,(\mathrm{T}_\sigma(u) \odot \Sigma_P); (\boldsymbol{\nu}u)\,(\mathrm{L}_\sigma(u) \mid P))\ \mathcal{R}(\Sigma_Q, Q)$.

The discreet bisimilarity relation $\approx_{\mathrm{R}}$ on $\mathbb{S}$ is the largest discreet bisimulation.

Point 1 is a standard requirement of a weak bisimulation but only tests plain and linear channels.

Point 2 says the following, for any available ω-server:

- The process remains discreet after having accepted an ω-request.
- If a decoding request is accepted then it has to be replied deterministically.
- Data decoding has no side effect and is uniform.
- Both processes provide the same data on a given ω-name.

Finally, point 3 tests ω-requests to the environment. We use template processes to ensure that all requests on the same ω-name receive the same answer, to allow processes like $\overline{u}(\boldsymbol{\nu}x).(P \mid Q)$ and $\overline{u}(\boldsymbol{\nu}x).P \mid \overline{u}(\boldsymbol{\nu}x).Q$ to be bisimilar. Note that ω-requests by themselves being without side effects, whether a request is sent or not has no effect on bisimilarity, for instance we have $(u : \uparrow_\omega; \overline{u}) \approx_{\mathrm{R}} (\emptyset; \mathbf{0})$.

Lemma 2.6.2 *Let relations $\mathcal{R}_1$ and $\mathcal{R}_2$ be discreet bisimulations.*

Then both $\mathcal{R}_\sigma = \mathcal{R}_1 \cup \mathcal{R}_2$ and $\mathcal{R}_\pi = \mathcal{R}_1\mathcal{R}_2 \cup \mathcal{R}_2\mathcal{R}_1$ are discreet bisimulations as well.

Note that R_1 and R_2 being bisimulations usually does not imply that $\mathcal{R}_1\mathcal{R}_2$ is, because symmetry is usually not preserved by composition.

It is however not a bisimilarity in the strict sense of the term because it is not reflexive — there are some processes that are not bisimilar to any other processes, not even themselves. This prompts for the following:

Definition 2.6.3 (Discreetness) *P is said* Σ-discreet *if* $(\Sigma; P) \approx_{\mathrm{R}} (\Sigma; P)$.

On Σ-discreet processes, $\approx_{\mathrm{R}}$ is an equivalence relation. $P \approx Q$ implies $(\Sigma; P) \approx_{\mathrm{R}} (\Sigma; Q)$ but the opposite direction usually does not hold. Consider the following pair (a, w and w' are plain, x is ω and both v and v' are linear)

$P = a(x).\,\big(\overline{x}(\boldsymbol{\nu} v).v(w).\overline{w} \mid \overline{x}(\boldsymbol{\nu} v).v'(w').\overline{w'}\big)$,

$Q = a(x).\,\overline{x}(\boldsymbol{\nu} v).v(w).(\overline{w}|\overline{w})$.

Consider the following transitions, available in sequence from P:

$\xrightarrow{a(b)} \xrightarrow{\overline{b}(\boldsymbol{\nu} v)} \xrightarrow{v(w_1)} \xrightarrow{\overline{b}(\boldsymbol{\nu} v')} \xrightarrow{v'(w_2)} \xrightarrow{\overline{w_1}} \xrightarrow{\overline{w_2}}.$

Plain outputs observed at the end may differ in P, but must be identical in Q:

$\xrightarrow{a(b)} \xrightarrow{\overline{b}(\boldsymbol{\nu} v)} \xrightarrow{v(w_1)} \xrightarrow{\overline{w_1}} \xrightarrow{\overline{w_1}}$

No such difference can be observed with discreet bisimulation, as ω-requests are not tested directly but by inserting a template process.

In the following lemma, a *context type* is an expression generated by the following grammar, Σ being a regular type mapping and x a name.

$K \ ::= \ [\cdot] \ \big| \ K \cdot \Sigma \ \big| \ K \setminus \Sigma \ \big| \ (\boldsymbol{\nu} x)\, K.$

Such a context type is typically checked against a process context $C[\cdot]$ starting from $([\cdot]; [\cdot])$ as the most basic typed context, and then following the rules given in Definition 2.3.1.

Lemma 2.6.4 (Bisimulation Congruence) *Let* $(\Sigma_P, P) \approx_{\mathrm{R}} (\Sigma_Q, Q)$.

If $(\Sigma[\cdot]; C[\cdot])$ *is a typed context s.t. both* $(\Sigma[\Sigma_P], C[P])$ *and* $(\Sigma[\Sigma_Q], C[Q])$ *are discreet, then* $(\Sigma[\Sigma_P], C[P]) \approx_{\mathrm{R}} (\Sigma[\Sigma_Q], C[Q])$.

Examples

We show how some non-discreet processes (failing the requirements given in introduction) are not discreet bisimilar with themselves.

They will all be processes of the form $P = \overline{a}\langle u\rangle.P'$ where P' listens on u (i.e. u should be thought as the encoding of the value transmitted over a). We have $P \xrightarrow{\overline{a}\langle u\rangle} P'$.

In the following, $p \neq q$, p, q, a, s have mode p, l has a linear mode and u has mode $\downarrow_\omega$. The examples are numbered to match section 1's requirements.

1. "*Once sent, the transmitted data is fully determined.*"

   $P' = (\boldsymbol{\nu} t)\,(\overline{t}|t.!\,u(l).\overline{l}\langle p\rangle|t.!\,u(l).\overline{l}\langle q\rangle)$

   As $P' \xrightarrow{u(x)}\xrightarrow{\overline{x}\langle p\rangle}$, $\Omega^{\Sigma}_{P'}(u) = \langle p\rangle$. We also have $P' \xrightarrow{u(x)}\xrightarrow{\overline{x}\langle q\rangle}$, so $\Omega^{\Sigma}_{P'}(u) = \langle q\rangle \neq \langle p\rangle$, which breaks last point of Definition 2.6.1's item 2. So $P' \not\approx_{\mathrm{R}} P'$, and (by item 1) $P \not\approx_{\mathrm{R}} P$.

2. "*the receiver can access it arbitrarily often and always gets the same result*"

   $P' = u(l).(\overline{l}\langle p\rangle \mid !\,u(l).\overline{l}\langle q\rangle)$

   We have $P' \xrightarrow{u(x)} P'' = \overline{x}\langle p\rangle \mid !\,u(l).\overline{l}\langle q\rangle$.

   Because $P'' \xrightarrow{\overline{x}\langle p\rangle}$, $\Omega^{\Sigma}_{P'}(u) = \langle p\rangle$, and because $(\boldsymbol{\nu} x)\,P'' \xrightarrow{u(y)}\xrightarrow{\overline{y}\langle q\rangle}$, $\Omega^{\Sigma}_{(\boldsymbol{\nu} x)\,P''}(u) = \langle q\rangle$. So $P' \not\approx_{\mathrm{R}} (\boldsymbol{\nu} x)\,P''$, which breaks the second point of requirement 2, so $P \not\approx_{\mathrm{R}} P$.

3. "*The sender can't know when and how many times the data is accessed*"

   $P' = !\,u(l).(\overline{l}\langle p\rangle \mid \overline{s})$

   This example sends a signal at each decoding request. $P' \xrightarrow{u(x)} P'' = !\,u(l).(\overline{l}\langle p\rangle \mid \overline{s}) \mid \overline{x}\langle p\rangle \mid \overline{s}$. As before, $P' \not\approx_{\mathrm{R}} P''$:

   $(\boldsymbol{\nu} l)\,P'' \xrightarrow{\overline{s}}$ but not $P' \xrightarrow{\overline{s}}$. so $P \not\approx_{\mathrm{R}} P$.

4. "*A decoding request is always replied to after a finite time*"

   $P' = !\,u(l).\mathbf{0}$

   There is no z such that $P \xrightarrow{\overline{a}\langle u\rangle} P' \xrightarrow{u(x)}\xRightarrow{\overline{x}\langle z\rangle}$, so $\Omega^{\Sigma}_{P'}(u)$ is undefined (which breaks second point of requirement 2), and $P \not\approx_{\mathrm{R}} P$.

3 Type System

In this section we propose a type system that analyses the behaviour of a process.

We will first define various formalisms and operators that will be required by the type system.

3.1 Receptiveness and Responsiveness

We distinguish receptiveness and responsiveness. The former stands for availability of input (or output) at a name, but provides no guarantees as to requests being replied.

A channel is said *input responsive* if the answer to a request depends at most on the request parameters. It is *output responsive* if the request's parameters depend at most on the reply. Output responsiveness only makes sense for linear channels, for which precisely one request is expected.

For example, in $\mathbf{!}\, a(x).\bar{b}.\bar{x}$ a is receptive but depends on b's receptiveness to be responsive.

3.2 Protocols

In a communication, there can be various conventions relating to which end should provide its parameters first. Examples include:

- *Anarchy* — No assumption can be done concerning the other end, which means that we may not make our parameters depend on the remote side.
- *Input First* — the input side must provide receptiveness on all its parameters without depending on receptiveness from the output side. Recursively, parameters follow the same convention. For instance, in $\overline{u}\langle x, y\rangle.\mathbf{!}\, y(t, u).P$, y must become receptive without depending on x (being an output parameter), however it may wait for the other end to become responsive before providing its own resources at t and u, as y itself is an input.
- *Left to Right* — Receptiveness or responsiveness on a parameter may only depend on parameters on its left.

Instead of choosing one particular convention, we decided to leave it open, and allow encoding any of them as a *protocol*, which is defined using dependencies between parameters. Then, when typing a process, we associate protocols to channel types.

A protocol is defined using dependencies between resources related to the parameters.

A *parameter resource* r is one of $i^{\downarrow}$, $i^{\uparrow}$,$\check{\imath}$ or $\hat{\imath}$, where i is a number specifying which parameter is being talked about. $\bar{r}$ is defined as follows. $\overline{i^{\downarrow}} = i^{\uparrow}$, $\overline{\check{\imath}} = \hat{\imath}$, $\overline{i^{\uparrow}} = i^{\downarrow}$ and $\overline{\hat{\imath}} = \check{\imath}$

Definition 3.2.1 (Protocol) *A protocol ρ for a type $(\sigma_1 \cdots \sigma_n)^m$ is an irreflexive transitive relation written $<_\rho$ on $\{i^{\downarrow}, i^{\uparrow}, \check{\imath}, \hat{\imath} : 1 \leq i \leq n\}$, and a sequence $\rho_1 \ldots \rho_n$, where ρ_i is a protocol for type σ_i called ρ's* sub-protocol *for parameter i.*

The complement *protocol $\bar{\rho}$ is s.t. $r_1 <_\rho r_2 \iff \overline{r_1} <_{\bar{\rho}} \overline{r_2}$ and $(\bar{\rho})_i = \rho_i$.*

$r_1 <_\rho r_2$ means the end providing r_2 may wait for r_1 before providing it.

More precisely:

Consider a communication $a(\tilde{y}).I \mid \overline{a}(\boldsymbol{\nu}\tilde{x}).O$.

Parameter resources with the up arrow indicate those (which may or may not be outputs) that O is expected to provide and those with the down arrow must be provided by I.

When checking compliance of a process to a communication protocol ρ, we model the behaviour of the other end assuming it is going to provide its resources as late as allowed by the protocol. In other words $r_1 \not<_\rho r_2$ means r_2's provider *must not* depend on r_1 for it.

This guarantees that the composition of two protocol-compliant processes is not going to generate deadlocks. If, instead, we had considered the remote side to provide resources as early as allowed by the protocol, a weakly-defined protocol (such as the Anarchy described above) would allow any behaviour both on input and on output side, which, when combined, may create a deadlock.

The complement protocol does has the same sub-protocols for the same reason the complement of a channel type has the same parameter types: because like channel types a protocol is by convention at the input's point of view, and the complement is the same but from output's point of view. In both cases description of the parameters is always from the input's point of view.

Examples

Consider the type $\sigma = (\downarrow_1, \uparrow_1)^{\downarrow_1}$.

As the parameters don't themselves carry parameters, the two sub-protocols will always be empty.

We use the following notation to describe orderings: R and S being resource sets, $R \mapsto S$ is the smallest relation $<$ such that $\forall r \in R : \forall s \in S : r < s$.

The corresponding anarchic protocol ρ^a is such that $<_{\rho^a} = \emptyset$, i.e. an empty relation expressing the lack of guarantees on the remote side's behaviour.

The "input fully responsive first" protocol ρ^i is defined such that $<_{\rho^i} = \{1^\downarrow, \check{1}, 2^\downarrow, \check{2}\} \mapsto \{1^\uparrow, \hat{1}, 2^\uparrow, \hat{2}\}$.

The following protocol requires all parameters to be fully receptive, and then to become fully responsive. It is interesting for modelling dialogue-type interactions, where each end does one step and then waits for the remote end to provide the next step, before continuing. ρ^s is such that $<_{\rho^s} = \{1^\downarrow, 1^\uparrow, 2^\downarrow, 2^\uparrow\} \mapsto \{\check{1}, \hat{1}, \check{2}, \hat{2}\}$

3.3 Action Types

Action types are defined following [YBH04], as a set of channel types together with causality information. We don't think as a graph but rather as a set of dependencies on resources but this point of view is equivalent.

Definition 3.3.1 (Resources) *A* resource *is one of x, $\check{x}$, $\hat{x}$, $(\tilde{x})$, where x is a name and $\tilde{x}$ a set of names.*

A resource is something that can be depended on. The four above cases are respectively receptiveness (both input and output) at x, input responsiveness, output responsiveness, and a *dependency group.* The exact semantics of dependency groups will be discussed when introducing action type composition.

Definition 3.3.2 (Extended Channel Type) *Let σ be a channel type, ρ be a protocol, $\tilde{\alpha}$, $\tilde{\beta}$ and $\tilde{\gamma}$ be possibly empty sets of resources respectively called the* receptive, input responsive *and* output responsive *conditions.*

An Extended Channel Type is a structure of the following form:

$$(\sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$$

When σ has a plain mode, all conditions are empty sets, and when σ has an ω-mode, output responsive conditions is an empty set.

This structure describes what a process provides at a name, and under what conditions. For instance the name is receptive once $\tilde{\alpha}$ are all available, and (provided σ is an input) is input responsive once $\tilde{\beta}$ are available. We omit the last components of the type if they are empty. For instance $(\sigma, \rho, \tilde{\alpha})$ stands for $(\sigma, \rho, \tilde{\alpha}, \emptyset, \emptyset)$.

Definition 3.3.3 (Action Type) *An* Action Type *is a mapping of names to extended channel types.*

We define for an action type A the following functions:

- The *channel type mapping*:
  $\Sigma_A(a) = \sigma$ iff $\exists \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma} \mid a : (\sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma}) \in A$.
  When $a \notin \mathrm{dom}(A)$, $\Sigma_A(a) \overset{\text{def}}{=} \epsilon$.
- *Dependency graph*: $\mapsto_A$ is a relation between resources based on names in $\mathrm{dom}(A)$, and such that, having $A(x) = (\sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$, $(z \mapsto_A x)$ if $z \in \tilde{\alpha}$, $(z \mapsto_A \check{x})$ if $z \in \tilde{\beta}$ and $(z \mapsto_A \hat{x})$ if $z \in \tilde{\gamma}$. Also, $r \mapsto_A \beta$ implies $r \mapsto_A (\tilde{x})$ if $\mathrm{n}(\beta) \in \tilde{x}$.

There are two Action Type composition operators, a simple merging operator $A + B$ and the actual composition operator $A \odot B$.

Merging is used when explicitly building an action type, where we do not need (or want) dependency tracking.

Definition 3.3.4 (Action Type Merging) *The* merging *of two action types A_1 and A_2, written $A_1 + A_2$, is defined as follows.*

Let $X = A_1 + A_2$. $\mathrm{dom}(X) = \mathrm{dom}(A_1) \cup \mathrm{dom}(A_2)$.

Assuming $A_i(x) = (\sigma_i, \rho, \tilde{\alpha}_i, \tilde{\beta}_i, \tilde{\gamma}_i)$ $(i = 1, 2)$,

$X(x) = (\sigma_1 \odot \sigma_2, \rho, \tilde{\alpha}_1 \cup \tilde{\alpha}_2, \tilde{\beta}_1 \cup \tilde{\beta}_2, \tilde{\gamma}_1 \cup \tilde{\gamma}_2)$.

The composition is not defined if one of the above is not defined (i.e. if a given name is in both A_i with uncomposable channel types)

$\sum_i T_i \overset{\text{def}}{=} T_1 + T_2 + \cdots$, *with the convention that* $\sum_{i \in \emptyset} T_i = \emptyset$.

We will now lift the definition of $\odot$ to Action Types, where it is (amongst others) used when parallel-composing two processes.

Definition 3.3.5 (Implied Dependency) *In the following, $A_{(i)}$ means A_1 if i is odd and A_2 otherwise.*

A pair $q \mapsto r$ is an implied dependency between two action types A_1 and A_2 iff there is a sequence $q = r_1, r_2 \ldots, r_n = r$ $(n > 2)$ s.t. both

1. $(\forall i : r_i \mapsto_{A_{(i)}} r_{i+1})$ *or* $(\forall i : r_{i-1} \mapsto_{A_{(i)}} r_i)$

2. *Let $\tilde{z}_i = \tilde{a}$ when r_i is of the form $(\tilde{a})$ and $\emptyset$ otherwise. Then $(\bigcup_i \tilde{z}_i) \cap (\bigcup_i \mathrm{n}(r_i)) = \emptyset$*

Note that a dependency entirely contained in either of the two action types is in general not an implied dependency because of the constraint $n > 2$.

Definition 3.3.6 (Action Composition) *The Action Type composition of two action types A_1 and A_2, written $A_1 \odot A_2$, is defined if there is no r such that $r \mapsto r$ is an implied dependency of A_1 and A_2, and if $A_1 \odot A_2$ is defined. Then,*

$A_1 \odot A_2$ is equal to $A_1 + A_2$, to which are added *dependencies that are* implied *by A_1 and A_2*

Note that we only check for circularities on implied dependencies, which means that composition only fails when it *creates* a circular dependency, not if one was already present in either action types.

Precedence: $A + B \odot C = (A + B) \odot C$ and $A \odot B + C = A \odot (B + C)$.

The process constructors such as repliction, restriction and prefixing are adapted to action types as operators.

Definition 3.3.7 (Action Type Replication) *$\mathbf{!}\, A$ is defined if and only if all names in A have a mode in $\{\downarrow_{\omega_0}, \uparrow_\omega, \mathtt{p}\}$, and[1] $\mathbf{md}(\Sigma_A\,(x)) = \downarrow_{\omega_0} \Rightarrow \nexists r : r \mapsto_A x$. Then:*

$\mathrm{dom}(\mathbf{!}\, A) \overset{\mathrm{def}}{=} \mathrm{dom}(A)$, *and assuming $A(a) = (\sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$, we have $(\mathbf{!}\, A)(a) = (\mathbf{!}\, \sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$, where*

$\mathbf{!}\ (\tilde{\sigma})^{\downarrow_{\omega_0}} \overset{\mathrm{def}}{=} (\tilde{\sigma})^{\downarrow_\omega}$, $\mathbf{!}\ (\tilde{\sigma})^{\mathtt{p}} \overset{\mathrm{def}}{=} (\tilde{\sigma})^{\mathtt{p}}$ *and* $\mathbf{!}\ (\tilde{\sigma})^{\uparrow_\omega} \overset{\mathrm{def}}{=} (\tilde{\sigma})^{\uparrow_\omega}$.

Similarly to [YBH04] we put some conditions on restricting names: for restricting a linear name we require to be both input and output, and for restricting an ω-name we require it to be listened in the restriction.

Definition 3.3.8 (Action Type Restriction) *Restricting a name x in an action type (written $(\boldsymbol{\nu} x)\, A$) is defined if and only if $\mathbf{md}(\Sigma_A\,(x)) \in m^{\bigstar}$, and when defined removes all x from the action type. Using $X = \{x, \check{x}, \hat{x}\}$,*

$\mathrm{dom}((\boldsymbol{\nu} x)\, A) \overset{\mathrm{def}}{=} \mathrm{dom}(A) \setminus x$.

$A(a) = (\sigma, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$ $(a \neq x)$ implies $((\boldsymbol{\nu} x)\, A)(a) = (\sigma, \rho, \tilde{\alpha} \setminus X, \tilde{\beta} \setminus X, \tilde{\gamma} \setminus X)$.

[1] Allowing conditions for $\downarrow_\omega$ would break subject reduction because though $\mathbf{!}\, P | \mathbf{!}\, P \sim \mathbf{!}\, P$ we do not allow more than one input on an ω-name. Allowing conditions for $\downarrow_{\omega_0}$ would allow "guarded uniformity", i.e. processes like $\mathbf{!}\, a.b$ where a and b would both be ω

Definition 3.3.9 (Action Type Prefixing) *Prefixing an action type A with a resource r of type σ (written $(r : \sigma).A$ or just $r.A$ if the type is unimportant or clear from context) is defined as follows.*

$\mathrm{dom}(r.A) = \mathrm{dom}(A)$

Let $A(a) = (\sigma_a, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$.

If $\sigma_a = (\tilde{\sigma})^{\downarrow_\omega}$ *and* $\mathbf{md}(\sigma) = \mathtt{p}$ *then* $\sigma'_a = (\tilde{\sigma})^{\bigstar_\omega}$. *Otherwise* $\sigma'_a = \sigma_a$.

If both $\mathbf{md}(\sigma_a)$ *and* $\mathbf{md}(\sigma)$ *are in* m^{r}, *then* $\tilde{\alpha}' = \{r\} \cup \tilde{\alpha}$, *otherwise* $\tilde{\alpha}' = \tilde{\alpha}$.

Then, $((r : \sigma).A)(a) \stackrel{\mathrm{def}}{=} (\sigma'_a, \rho, \tilde{\alpha}', \tilde{\beta}, \tilde{\gamma})$.

Notation: $abc.A$ stands for $a.b.c.A$.

Plain-prefixing an ω-input makes it uncomposable because in a process like $\overline{a}\langle u\rangle.\mathbf{!}\, u$, u is not guaranteed to ever become available and no request to u may be sent from outside the plain prefix.

The side conditions are needed to ensure non-linear resources neither depend on a resource, nor are depended upon.

The type system proposed at next section is a *weak* one, in the sense that a given process can be typed in different ways. This is needed to have a strong safety theorem, i.e. if a A types a process P then any reduction P' of P (i.e. $P \Rightarrow P'$) can also be typed by A.

Through reduction, dependencies due to over-approximation disappear, inert resources get consumed and ω-inputs marked uncomposable become composable if the plain prefix is consumed. Weakening allows reverting these changes.

Definition 3.3.10 (Action Type Weakening) *We say A' is a* weakening *of A ($A \leq A'$) if* $\mathrm{dom}(A) \subseteq \mathrm{dom}(A')$, *and:*

- $\forall a \in \mathrm{dom}(A)$, *let* $A(a) = ((\tilde{\sigma})^{m}, \rho, \tilde{\alpha}, \tilde{\beta}, \tilde{\gamma})$. *Then* $A'(a) = ((\tilde{\sigma})^{m'}, \rho, \tilde{\alpha}' \cup \tilde{\alpha}, \tilde{\beta}' \cup \tilde{\beta}, \tilde{\gamma}' \cup \tilde{\gamma})$ *s.t. either* $(m = m')$ *or* $(m = \downarrow_\omega \wedge m' = \bigstar_\omega)$. *As usual,* $\tilde{\alpha}'$, $\tilde{\beta}'$ *and* $\tilde{\gamma}'$ *must all be linear resources.*
- $\forall a \in \mathrm{dom}(A') \setminus \mathrm{dom}(A)$: $\mathbf{md}(\Sigma_{A'}(a)) \in m^{\bigstar}$.

We may now define *instantiation* of a protocol. Just as a parameter types in a channel type refer to the input's point of view (as to what the output is expected to provide), a protocol instantiation provides the input's point of view as to what the output may require to provide its parameters. Instantiating the protocol for the output's point of view is done on the complement protocol $\bar{\rho}$.

Definition 3.3.11 (Protocol Instantiation) *The* instantiation *of a protocol ρ with parameters $\tilde{x}$ of type $\tilde{\sigma}$, written $\rho(\tilde{x} : \tilde{\sigma})$, gives the action type defined as follows.*

In the following, $x'_j = \begin{cases} \check{x}_j & \text{if } \mathbf{md}(\sigma_j) \in \{\downarrow_1, \downarrow_\omega\} \\ \hat{x}_j & \text{otherwise} \end{cases}$.

$\rho(\tilde{x} : \tilde{\sigma}) \overset{\text{def}}{=} \sum_i x_i : (\sigma_i, \rho_i, \tilde{\alpha}_i, \tilde{\beta}_i, \tilde{\gamma}_i)$ *where:*
If $\mathbf{md}(\sigma_i) \in \{\mathtt{p}, \uparrow_\omega\}$ *then* $\tilde{\alpha}_i = \tilde{\beta}_i = \gamma_i = \emptyset$.
Otherwise, having $j \neq i$*:*
$x_j \in \tilde{\alpha}_i \iff i^\uparrow \not<_\rho j^\downarrow$
$x'_j \in \tilde{\alpha}_i \iff i^\uparrow \not<_\rho \hat{\jmath}$
$x_j \in \tilde{\delta}_i \iff \check{\imath} \not<_\rho j^\downarrow$
$x'_j \in \tilde{\delta}_i \iff \check{\imath} \not<_\rho \hat{\jmath}$
If $\mathbf{md}(\sigma_i) \in \{\downarrow_1, \downarrow_\omega\}$ *then* $\tilde{\beta}_i = \tilde{\delta}_i$ *and* $\tilde{\gamma}_i = \emptyset$.
If $\mathbf{md}(\sigma_i) = \uparrow_1$ *then* $\tilde{\beta}_i = \emptyset$ *and* $\tilde{\gamma}_i = \tilde{\delta}_i$.
Instantiation is undefined in case it results in a resource depending on itself.

Just as a parameter types in a channel type refer to the input's point of view (as to what the output is expected to provide), a protocol instantiation provides the input's point of view as to what the output may require to provide its parameters. Instantiating the protocol for the output's point of view is done on the symmetric protocol $\bar{\rho}$.

Examples

As before we work on the type $\sigma = (\downarrow_1, \uparrow_1)^{\downarrow_1}$.

As a first example, let ρ^a be the anarchic (empty) protocol.

Then, $\rho^a(x : \downarrow_1, y : \uparrow_1) = \{x : (\downarrow_1, \emptyset, y\hat{y}, y\hat{y}); y : (\uparrow_1, \emptyset, x\check{x}, \emptyset, x\check{x})\}$

Because $r_1 \not<_{\rho^a} r_2$ for any r_1 and r_2, all possible dependencies are put in the resulting action type. The apparent circular dependencies (for instance we have $x \mapsto y \mapsto x$) are put in place to prevent composing with any local inter-parameter dependency. More specifically, the action type will compose with a type of process $\bar{x}|y$ but not with either $\bar{x}.y$ or $y.\bar{x}$.

If in this example, $y = x$, however, the instantiation is no longer defined because it would result to $\rho^a(x : \downarrow_1, x : \uparrow_1) = E = \{x : (\bigstar_1, \emptyset, x\hat{x}\check{x}, x\hat{x}\check{x})\}$ where for instance $x \mapsto_E x$.

If we use the "output fully responsive first" protocol, we simply get $\rho^o(x : \downarrow_1, y : \uparrow_1) = \{x : \downarrow_1; y : \uparrow_1\}$. The remote parameters are created with no dependencies, to allow the local (input) resources to fully depend on the remote one.

$$\frac{-}{\emptyset \vdash \mathbf{0}}\ (\text{Nil}) \qquad \frac{A \vdash_\pi P \quad A \le A'}{A' \vdash_\pi P \quad A' \vdash_{\mathrm{p}} P}\ (\text{Weak}) \qquad \frac{A \vdash_\pi P}{!\,A \vdash_\pi !\,P}\ (\text{Rep})$$

$$\frac{i = 1,2 : A_i \vdash_\pi P_i}{A_1 \odot A_2 \vdash_\pi P_1 \mid P_2}\ (\text{Par}) \qquad \frac{A \vdash_\pi P}{(\boldsymbol{\nu} x)\, A \vdash_\pi (\boldsymbol{\nu} x)\, P}\ (\text{Res})$$

Table 2: Type System — Basic Rules

One last example, consider the protocol "parameters made available from left to right" $\rho^{\rightarrow}$, based on $\{1^{\downarrow}, 1^{\uparrow}, \check{1}, \hat{1}\} \mapsto \{2^{\downarrow}, 2^{\uparrow}, \check{2}, \hat{2}\}$. Here the instantiation gives $\rho^{\rightarrow}(x :\downarrow_{\mathtt{l}}, y :\uparrow_{\mathtt{l}}) = \{x :\downarrow_{\mathtt{l}}; y : (\uparrow_{\mathtt{l}}, \emptyset, x\check{x}, \emptyset, x\check{x})\}$.

The $x \mapsto y$ dependencies are put in place to prevent the local resources to have a dependency such as $y \mapsto x$, so that an input continuation such as $\bar{x}|y$ or $\bar{x}.y$ would be accepted, but not $y.\bar{x}$.

3.4 Determinism rules

In addition to relating a process to an action type, typing judgements also specify whether a process might use a plain name as a channel. The reason why we don't express this as an action mode is that this information is needed even if the plain name is bound, as in $l(p).\bar{p}$.

$A \vdash P$ means P behaves according to the action type A, and *will not* use any plain name as a channel.

$A \vdash_{\mathrm{p}} P$ means P behaves according to the action type A, and *may* use a plain name as a channel.

$A \vdash_\pi P$ means $\pi = \mathtt{p} \Rightarrow A \vdash_{\mathrm{p}} P$, and $\pi \neq \mathtt{p} \Rightarrow A \vdash P$.

We now have all the material to write the rules of the type system, given in Tables 2 and 3.

The weakening rule allows either weakening just the action type and keeping π, or also replacing a $A \vdash P$ by $A \vdash_{\mathrm{p}} P$.

Composing $A \vdash P$ and $B \vdash_{\mathrm{p}} Q$ with the parallel composition rule is done by first (WEAK)-ening $A \vdash P$ to $A \vdash_{\mathrm{p}} P$ (this is required by (Par) to have a single π value)

All prefix rules are essentially the same.

We first state that the channel is immediately receptive, and for responsiveness depends on what its parameters depend.

Then we add the remote parameters with the $+$ operator by instantiating the protocol. The reason for not using $\odot$ is that it would create fake dependencies. For example, consider a linear input $l(x).P$. l's type is set

$$\frac{A \vdash_\pi P \qquad \forall l : \mathbf{md}(\Sigma_A(l)) \notin \{\downarrow_{\mathtt{1}}, \uparrow_{\mathtt{1}}, \downarrow_{\omega_0}\}}{p.(\boldsymbol{\nu}\tilde{x})\ (p : ((\tilde{\sigma})^{\mathtt{p}}, \rho) + \rho(\tilde{x} : \tilde{\sigma}) \odot A) \vdash_{\mathrm{p}} p(\tilde{x}).P}\ (\textsc{Inp}_{\mathtt{p}})$$

$$\frac{A \vdash_\pi P}{(\boldsymbol{\nu}\tilde{x})\ \left(l : ((\tilde{\sigma})^{\downarrow_{\mathtt{1}}}, \rho, \emptyset, (\tilde{x})) + l\hat{l}.\rho(\tilde{x} : \tilde{\sigma}) \odot l.A\right) \vdash_\pi l(\tilde{x}).P}\ (\textsc{Inp}_{\mathtt{1}})$$

$$\frac{A \vdash\ P}{(\boldsymbol{\nu}\tilde{x})\ \left(u : ((\tilde{\sigma})^{\downarrow_{\omega_0}}, \rho, \emptyset, (\tilde{x})) + u.\rho(\tilde{x} : \tilde{\sigma}) \odot u.A\right) \vdash\ u(\tilde{x}).P}\ (\textsc{Inp}_{\omega})$$

$$\frac{A \vdash_\pi P \quad \uparrow_\omega \notin \mathbf{md}(\tilde{\sigma}) \quad \forall l : \mathbf{md}(\Sigma_A(l)) \notin \{\downarrow_{\mathtt{1}}, \uparrow_{\mathtt{1}}, \downarrow_{\omega_0}\}}{p.\left(p : ((\overline{\tilde{\sigma}})^{\mathtt{p}}, \rho) + \bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}}) \odot A\right) \vdash_{\mathrm{p}} \overline{p}\langle\tilde{x}\rangle.P}\ (\textsc{Out}_{\mathtt{p}})$$

$$\frac{A \vdash_\pi P}{l : ((\overline{\tilde{\sigma}})^{\uparrow_{\mathtt{1}}}, \rho, \emptyset, \emptyset, (\tilde{x})) + l\check{l}.\bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}}) \odot l.A \vdash_\pi \overline{l}\langle\tilde{x}\rangle.P}\ (\textsc{Out}_{\mathtt{1}})$$

$$\frac{A \vdash_\pi P}{u : ((\tilde{\sigma})^{\uparrow_\omega}, \rho) + u\check{u}.\bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}}) \odot u.A \vdash_\pi \overline{u}\langle\tilde{x}\rangle.P}\ (\textsc{Out}_{\omega})$$

Table 3: Type System — Input/Output

to have $(x) \mapsto \check{l}$. Then, protocol instantiation will add a dependency $\hat{l} \mapsto x$ (because the remote parameter depends on output responsiveness). If $\odot$ were used then input responsiveness would wrongly depend on output responsiveness. A similar situation would happen on the output side, and we would obtain a dependency loop like $\check{l} \mapsto \hat{l} \mapsto \check{l}$. Intuitively the idea is that local responsiveness should not be expanded with dependencies on remote parameters but only local ones, which is why it is not $\odot$-composed with protocol instantiation.

Finally continuation is composed with the resulting action type, after adding to it a dependency on receptiveness of the channel.

Restricting parameters in input rules has two purposes. First, it makes sure the parameters won't be visible in the resulting action type, as only complete modes can be restricted. Second, it makes sure the continuation A has provided required resources (e.g. for a linear parameter, one polar will be provided by the remote end and A must provide the other one so that they combine into $\bigstar_{\mathtt{1}}$ which can be restricted.)

Plain prefix rules force the resulting judgement to record a plain interaction, while ω-input requires the continuation not to contain any plain interaction.

Note that the action type prefixing on the channel is done outside for plain names. The reason is that when σ has mode $\mathtt{p}$, $((p:\sigma).A)\odot((p:\sigma).B)$ is in general not equal to $(p:\sigma).(A\odot B)$ in case there is a communication over an ω name happening between A and B. This is because plain prefixing changes $\downarrow_\omega$ modes into $\bigstar_\omega$, to express the fact that communicating with a plain prefixed ω input is not reliable and should be blocked. However, it is reliable when the ω name is a parameter, and the input is provided in the output continuation, as in $(p(x).\bar{x})\mid(\overline{p}\langle u\rangle.\mathbf{!}\,u)$. We do not put the prefixing operator on the outside in the other rules to avoid having the channel depend on itself.

The $\uparrow_\omega\notin\mathbf{md}(\tilde{\sigma})$ side condition of the plain output is used to prevent processes such as $(\boldsymbol{\nu}p)\ (p(x).\mathbf{!}\,x(y).\overline{y}\langle q\rangle\mid p(x).\mathbf{!}\,x(y).\overline{y}\langle q'\rangle\mid\overline{p}\langle u\rangle)$ which is not discreet as $\Omega_P(u)$ can be either $\langle\langle q\rangle\rangle$ or $\langle\langle q'\rangle\rangle$.

Parameters with linear modes are restricted for transmission over plain channels the following way:

$p(l)$, where p is plain and l either $\downarrow_1$ or $\uparrow_1$ can't be completed by the continuation, and the incomplete mode can't be restricted in the input rule.

$\overline{p}\langle u\rangle$, where u is $\uparrow_\omega$, is rejected, by the special side condition in (Out$_{\mathtt{p}}$).

Both $p(u)$ and $\overline{p}\langle u\rangle.P$, where u is $\downarrow_\omega$ is accepted, but the u name won't be usable outside of the prefix, as the communication is not guaranteed. This is enforced by the action type prefixing operator setting the mode to $\bigstar_\omega$.

The type system does allow processes like $\overline{p}\langle l\rangle\mid l$, but these may not be used in any way, as no typable context may provide a corresponding input. Same for processes like $p(u).\mathbf{!}\,u$.

Other linear interactions crossing the boundary of a plain prefix (on names different from the parameters) are restricted in the same way.

Plain interactions in ω-input continuations is prevented by the (Inp$_\omega$) requiring a "$A\ \vdash$ "-typing for the continuation.

3.5 Example

This is an example of an ω-server that provides a linear input at its parameter.

Consider $\mathbf{!}\,u(x).x\mid\overline{u}\langle l\rangle.\bar{l}$, where $\bar{l}$ abbreviates of $\bar{l}\langle\rangle.\mathbf{0}$. This process can be typed using $u:\sigma_u=\left(()^{\uparrow_1}\right)^{\downarrow_\omega}$ and $l:()^{\bigstar_1}$, and the ("anarchic") protocol $\rho=\emptyset$.

We start with (Nil): $\emptyset\ \vdash\ \mathbf{0}$, which is passed as a premise to (Out$_1$), yielding

$$l : ()^{\uparrow_1} + l\check{l}.\emptyset \odot l.\emptyset \;\vdash\; \bar{l}\langle\rangle.\emptyset$$

i.e. $l :\uparrow_1 \;\vdash\; \bar{l}$, which in turn is passed to (Out$_\omega$):

$$u : \left(()^{\uparrow_1}\right)^{\uparrow_\omega} + u\check{u}.\bar{\rho}(l :\downarrow_1) \odot u.l :\uparrow_1 \;\vdash\; \overline{u}\langle l\rangle.\bar{l}$$

Applying the empty protocol $\bar{\rho} = \emptyset$ to $l :\downarrow_1$ we get simply $l :\downarrow_1$. (As protocol instantiation only creates inter-parameter dependencies, the one-parameter case will never create any dependency.) Evaluating the prefixes and the $+$ operator:

$$\left\{u : \left(()^{\uparrow_1}\right)^{\uparrow_\omega}; l : (\downarrow_1, \emptyset, u\check{u})\right\} \odot l : (\uparrow_1, \emptyset, u) \;\vdash\; \overline{u}\langle l\rangle.\bar{l}$$

As there is no implied dependency, $\odot$ behaves as $+$. Note that $\downarrow_1$ and $\uparrow_1$ get composed into $\bigstar_1$. The shortest typing for this process is then

$$\left\{u : (\uparrow_1)^{\uparrow_\omega}; l : (\bigstar_1, \emptyset, u\check{u})\right\} \;\vdash\; \overline{u}\langle l\rangle.\bar{l} \qquad (1)$$

The dependencies on l show that for this channel to reach completeness we will need a responsive input at u. It is required for an input at l to be available.

The input x is typed using (Inp$_1$) similarly to $\bar{l}$: $x :\downarrow_1 \;\vdash\; x$.

We type the left hand side of our process using (Inp$_\omega$).

$$(\boldsymbol{\nu}x)\; \left(u : ((\uparrow_1)^{\downarrow_{\omega_0}}, \emptyset, \emptyset, (x)) + u.\rho(x :\uparrow_1) \odot u.x :\downarrow_1\right) \;\vdash\; u(x).x$$

Instantiating the protocol, applying the prefixes and the $+$.

$$(\boldsymbol{\nu}x)\; \left(\left\{u : ((\uparrow_1)^{\downarrow_{\omega_0}}, \emptyset, \emptyset, (x)); x : (\uparrow_1, \emptyset, u)\right\} \odot x : (\downarrow_1, \emptyset, u)\right) \;\vdash\; u(x).x$$

Applying the remaining $\odot$. As $(x) \mapsto \check{u}$ on the left side and $u \mapsto x$ on the right side, we get $u \mapsto \check{u}$ as an implied dependency:

$$(\boldsymbol{\nu}x)\; \left\{u : ((\uparrow_1)^{\downarrow_{\omega_0}}, \emptyset, \emptyset, (x)u); x : (\bigstar_1, \emptyset, u)\right\} \;\vdash\; u(x).x$$

Applying the restriction is well-defined because x has the complete mode $\bigstar_1$:

$$u : ((\uparrow_1)^{\downarrow_{\omega_0}}, \emptyset, \emptyset, u); \;\vdash\; u(x).x$$

The corresponding type for the replicated process is

$$u : ((\uparrow_1)^{\downarrow_\omega}, \emptyset, \emptyset, u);\ \vdash\ !\,u(x).x$$

which we can compose with (1), getting the following action type:

$$\Big\{u : ((\uparrow_1)^{\downarrow_\omega}, \emptyset, \emptyset, u)\Big\} \odot \Big\{u : (\uparrow_1)^{\uparrow_\omega}; l : (\bigstar_1, \emptyset, u\check{u})\Big\}$$

The composition operators defines $\downarrow_\omega \odot \uparrow_\omega = \downarrow_\omega$:

$$u : ((\uparrow_1)^{\downarrow_\omega}, \emptyset, \emptyset, u); l : (\bigstar_1, \emptyset, u\check{u})$$

All dependencies may be removed as u is complete, giving us the final typing

$$u : (\uparrow_1)^{\downarrow_\omega}, l : \bigstar_1\ \vdash\ !\,u(x).x \mid \overline{u}\langle l\rangle.\bar{l}$$

3.6 Properties

3.6.1 Convention

In all the following lemmas, properties and theorems we will assume the typing of processes mentioned in the statements do not contain any name of mode $\downarrow_{\omega_0}$, i.e. that each ω name has been replicated. It is indeed possible to build broken (but typable) processes such as $(u \mid \bar{u}.\bar{u}.l)$ where u is $\downarrow_{\omega_0}$ and l is $\downarrow_1$. This process is broken in that it can't be extended to have u become $\downarrow_\omega$, and in this form the l input can't happen. This is why none of the theorems apply to such a process.

We have the following properties.

Lemma 3.6.1 (Names) *If $A \vdash_\pi P$ then* $\mathrm{dom}(A) \supseteq \mathrm{fn}(P)$ *and* $\exists A' \leq A$ *with $A' \vdash_\pi P$ and* $\mathrm{dom}(A') = \mathrm{fn}(P)$.

The following Lemma expresses the fact that each typable process has one "strongest" type A'.

Lemma 3.6.2 (Strongest Type) *Let $A \vdash_\pi P$. Then there is A_0 s.t. $\forall A'$, $A' \vdash_\pi P$ iff $A_0 \leq A'$.*

Lemma 3.6.3 (Decidability) *The typability of any (finite) process P (i.e. whether $\exists A : A \vdash_{\mathrm{p}} P$) is decidable, and for any A, P and π, whether $A \vdash_\pi P$ holds is decidable as well.*

Moreover there is a linear time algorithm for determining the strongest type for any (finite) process.

The following safety proposition only tests transitions available both in the process and in the action type. Transitions unavailable in the action type would not be available in a typable environment, and would render the resulting process untypable as well (for instance $l(x).x \mid l' \xrightarrow{l(l')} l'|l'$).

Proposition 3.6.4 (Type Safety) *Let $A \vdash_\pi P$ and $(\Sigma_A; P) \xrightarrow{\mu} (\Sigma'; P)$. Then there is an action type A' s.t. $A' \vdash_\pi P'$ and $\Sigma_{A'} = \Sigma'$. If $\mu = \tau$ then $A = A'$.*

When outputting a bound name or inputing a name, it must appear in the resulting action type, with the correct channel type.

Lemma 3.6.5 (Type Soundness) *Let $A \vdash_\pi P$ and $a \in \mathrm{dom}(A)$ s.t. $\nexists r : r \mapsto_A a$. Let $\Sigma_A(a) = (\tilde{\sigma})^m$. Then:*

- *If $m = \uparrow_1$ then there is P' with $P \xrightarrow{(\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{x}\rangle} P'$*
- *If $m = \downarrow_1$ then there is P' with $P \xrightarrow{a(\tilde{x})} P'$*
- *If $m = \downarrow_\omega$ then there is P' with $P \xrightarrow{a(\tilde{x})} P'$*

The proof is in section 4.5.

Lemma 3.6.6 (Consistency) *If $A \vdash_\pi P$ then Σ_A is consistent for P.*

Proposition 3.6.7 (Discreet Soundness) *Let $A \vdash_\pi P$.
Then P is Σ_A-discreet.*

I.e. the type system is a characterisation of discreetness. The proof is in Section 4.8

4 Proofs

4.1 Notation

We introduce some additional notation that will be used for the proofs.

Template processes are generalised to type mappings:

$$\mathrm{L}(\Sigma) \stackrel{\text{def}}{=} \prod_{a \in \mathrm{dom}(\Sigma)} \mathrm{L}_{\Sigma(a)}(a)$$

4.2 Congruence

Let $(\Sigma_P, P) \approx_{\mathrm{R}} (\Sigma_Q, Q)$.

If $(\Sigma[\cdot]; C[\cdot])$ is a typed context s.t. both $(\Sigma[\Sigma_P], C[P])$ and $(\Sigma[\Sigma_Q], C[Q])$ are discreet, then $(\Sigma[\Sigma_P], C[P]) \approx_{\mathrm{R}} (\Sigma[\Sigma_Q], C[Q])$.

This is proven by induction on $C[\cdot]$

4.2.1 Bisimilarity Preservation (Lemma 2.6.2)

$\mathcal{R}_\sigma$ (Union):

Let $P\mathcal{R}_\sigma Q$. Then $P\mathcal{R}_i Q$ with $i = 1$ or $i = 2$. All conditions on $\mathcal{R}_\sigma$ (including symmetry) being existential (of the form "if $P \xrightarrow{\mu} P'$ then there are R and S such that $R\mathcal{R}_\sigma S$" with some conditions on R and S), they holding on $\mathcal{R}_i$ implies them holding on $\mathcal{R}_\sigma$ as well.

$\mathcal{R}_\pi$ (Composition):

Symmetry: Let $A\mathcal{R}_\pi B$. Then $A\mathcal{R}_1\mathcal{R}_2 B$ or $A\mathcal{R}_2\mathcal{R}_1 B$, i.e. $\exists X$ s.t. either $A\mathcal{R}_1 X$ and $X\mathcal{R}_2 B$, or $A\mathcal{R}_2 X$ and $X\mathcal{R}_1 B$.

By symmetry of both $\mathcal{R}_i$, we either have $B\mathcal{R}_2 X$ and $X\mathcal{R}_1 A$ or $B\mathcal{R}_1 X$ and $X\mathcal{R}_2 A$, i.e. $B\mathcal{R}_2\mathcal{R}_1 A$ or $B\mathcal{R}_1\mathcal{R}_2 A$. So $B\mathcal{R}_\pi A$.

We now show that, except symmetry, all conditions defining a bisimulation are preserved for both $\mathcal{R}_1\mathcal{R}_2$ and $\mathcal{R}_2\mathcal{R}_1$. The proof for $\mathcal{R}_\sigma$ can then be adapted for their union.

τ transitions are preserved by transitivity of $\Rightarrow$.

For $\mu \neq \tau$ transitions, $A \xrightarrow{\mu} A'$ is matched by the first bisimulation as $X \overset{\mu}{\Longrightarrow} X'$, which is matched by the second as $B \Rightarrow \overset{\mu}{\Longrightarrow} \Rightarrow B'$.

Observable value preservation is immediate, combining $\Omega_A(x) = \Omega_X(x)$ and $\Omega_X(x) = \Omega_B(x)$.

Other requirements apply on a single side of the relation at a time, i.e. are of the form "$P\mathcal{R}Q$ implies $pr(P)$ and (by symmetry) $pr(Q)$" (for some property pr). So, having $A\mathcal{R}_\pi B$ and let X be the intermediary process as above, we get $pr(A)$, $pr(X)$ and $pr(B)$.

4.3 τ-Safety

Action Types constructed by the type system are of the form $A_1 \odot A_2 \odot \cdots A_n$ where no $\odot$ operator has been used to construct the A_i.

Then, assuming $A = A_1 \odot \cdots \odot A_n$, we may define $A\{^{\tilde{x}}/_{\tilde{y}}\} = A_1\{^{\tilde{x}}/_{\tilde{y}}\} \odot \cdots \odot A_n\{^{\tilde{x}}/_{\tilde{y}}\}$ where substitution is defined on name-extended channel types the natural way and generalised to action types as follows[2].

[2]The sum is used to handle name collisions

$$A_i\{^{\tilde{x}}/_{\tilde{y}}\} \stackrel{\mathrm{def}}{=} \sum_{(a:r)\in A_i} \big((a:r)\{^{\tilde{x}}/_{\tilde{y}}\}\big)$$

Protocol instantiation models a remote behaviour. The next lemma states that if the remote side is typable then its type is equal (after weakening) to the protocol instantiation.

Lemma 4.3.1 (Protocol Instantiation is Faithful) *If $A = (\boldsymbol{\nu}\tilde{y})\,(B \odot \rho(\tilde{y} : \tilde{\sigma})) \odot \bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}})$ is well defined then $A \leq B\{^{\tilde{x}}/_{\tilde{y}}\}$.*

Proof

We map abstract resources to corresponding parameter resources as follows: $i^{\downarrow} \mapsto y_i$, $i^{\uparrow} \mapsto y_i$, $\check{\imath} \mapsto \check{y}_i$ and $\hat{\imath} \mapsto \hat{y}_i$. ρ's abstract resource partial ordering $<_\rho$ is thus translated to a relation on resources $<^{\tilde{y}}_\rho$. The subgraph of $\mapsto_B$ containing only resources based on names in $\tilde{y}$ is written $\mapsto_B\big|_{\tilde{y}}$.

Then, as $B \odot \rho(\tilde{y} : \tilde{\sigma})$ is well defined, $\mapsto_B\big|_{\tilde{y}} \subseteq <^{\tilde{y}}_\rho$.

□

Let $A \vdash_\pi P$ and $(\Sigma_A; P) \;\rightarrow (\Sigma_A; P')$. We want $A \vdash_\pi P'$.

We first handle a very basic case, and then generalise it to arbitrary processes. Safety is proven essentially the same way for plain, linear and ω-channels. We show the case of a plain communication.

Let $A \vdash_\pi P = a(\tilde{y}).R \mid \overline{a}\langle\tilde{x}\rangle$. Clearly, $(\Sigma_A; P) \;\rightarrow (\Sigma_A; R\{^{\tilde{x}}/_{\tilde{y}}\})$.

Let $B \vdash_{\pi'} R$ be the typing used when typing P.

Then $B\{^{\tilde{x}}/_{\tilde{y}}\} \vdash_{\pi'} R\{^{\tilde{x}}/_{\tilde{y}}\}$.

We now study how A is built from B.

$a.(\boldsymbol{\nu}\tilde{y})\;(a : ((\tilde{\sigma})^{\mathtt{P}}, \rho) + \rho(\tilde{y} : \tilde{\sigma}) \odot B) \vdash_{\mathrm{p}} a(\tilde{y}).R$

composed with

$a.\big(a : ((\overline{\tilde{\sigma}})^{\mathtt{P}}, \rho) + \bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}})\big) \vdash_{\mathrm{p}} \overline{a}\langle\tilde{x}\rangle$

yields

$a.(\boldsymbol{\nu}\tilde{y})\;\big(a : ((\tilde{\sigma})^{\mathtt{P}}, \rho) + \rho(\tilde{y} : \tilde{\sigma}) \odot B \odot \bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}})\big) \vdash_{\mathrm{p}} P.$

Removing the restricted names from the type:

$a : ((\tilde{\sigma})^{\mathtt{P}}, \rho) \odot a.\big(B \setminus \tilde{y} \odot \bar{\rho}(\tilde{x} : \overline{\tilde{\sigma}})\big) \vdash_{\mathrm{p}} P$

After transition, the type is $B\{^{\tilde{x}}/_{\tilde{y}}\} \vdash_\pi R\{^{\tilde{x}}/_{\tilde{y}}\} = P'$.

By weakening: $a : ((\tilde{\sigma})^{\mathtt{P}}, \rho) \odot a.(B\{^{\tilde{x}}/_{\tilde{y}}\}) \vdash_{\mathrm{p}} P'$.

By Lemma 4.3.1, $B\{^{\tilde{x}}/_{\tilde{y}}\}$ can be further weakened to be equal to P's typing.

This proof is easily generalised to arbitrary processes:

Let $A \vdash_\pi P \;\rightarrow P'$. Reorganising P's structure we can bring the input and output side of the communication together, and have $P \equiv C[P_0] \;\rightarrow$

$C[P_0'] \equiv P'$, where $A_0 \vdash_\pi P_0$ is of the above form. By the above proof, $A_0 \vdash_\pi P_0'$, and the development from $A_0 \vdash_\pi P_0$ to $A \vdash_\pi P$ can be applied to $A_0 \vdash_\pi P_0'$, yielding $A \vdash_\pi P'$

4.4 Safety, general case

Let $A \vdash_\pi P$.

Safety is generalised to arbitrary transitions as follows.

For $\mu = a(\tilde{x})$, consider the process $Q = P \mid \overline{a}\langle\tilde{x}\rangle$. It is typable if and only if the μ-transition is available on $(\Sigma_A; P)$ as they are both subject to the same side conditions on $\odot$. Then, the μ-transition from P leads to the same process as a τ-transition from Q, and τ-safety can be used for the latter.

For $\mu = (\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{x}\rangle$, a similar trick can be used, by composing with a corresponding input process. After reduction, the restrictions on names in $\tilde{z}$ can be removed, making the names visible in the type while keeping typability, because if a process is typable then so are its sub-processes.

4.5 Soundness

Let $A \vdash_\pi P$ and $a \in \mathrm{dom}(A)$ s.t. $\nexists r : r \mapsto_A a$. Let $\Sigma_A(a) = (\tilde{\sigma})^m$. Then:

- If $m = \uparrow_1$ then there is P' with $P \xRightarrow{(\boldsymbol{\nu}\tilde{z})\,\overline{a}\langle\tilde{x}\rangle} P'$
- If $m = \downarrow_1$ then there is P' with $P \xRightarrow{a(\tilde{x})} P'$
- If $m = \downarrow_\omega$ then there is P' with $P \xRightarrow{a(\tilde{x})} P'$

Defining the above three conditions as *availability* of the corresponding resource, we define a partial order $<_A$ on resources corresponding to the *non-reduced* dependency network, i.e. where dependencies are not forwarded or simplified.

Availability of active resources is then proven by induction on $<_A$.

4.6 Value Uniqueness

Lemma 4.6.1 (Value Uniqueness) *Let $A \vdash_\pi P$. Then $\forall a \in \mathrm{dom}(A)$, $\exists! \xi : \Omega_P(a) = \xi$.*

Proof By induction on the process structure, by computing $\Omega_P^{\Sigma_A}$. Prefixing processes groups observable values of the continuation into a record, while parallel composition substitutes indexed data components being provided by a term of the parallel composition.

The side conditions of the $\odot$ operator allows to guarantee uniqueness, while the side conditions of the restriction operator (used by the prefix rules) guarantee existence.

$\square$

4.7 Closeness

Lemma 4.7.1 (Closeness) *Let $A \vdash P$. Then $P \approx_{\mathrm{R}} \mathrm{L}(\Sigma_A)$, up to α-renaming.*

Proof Again, by induction on P's structure, by showing it is bisimilar to the product of template processes instantiated with P's observable data. Interactions on plain channels are prevented as they would produce a typing like $A \vdash_{\mathrm{p}} P$. $\square$

4.8 Discreetness

The following definition characterises ω-input continuations whose reply channels are bound. Note that an inert process is not necessarily deadlocked. One example of inert process would be $(u :\uparrow_\omega ; \bar{u})$.

Definition 4.8.1 (Inert Process) *A typed process $(\Sigma; P)$ is said* inert *if $(\Sigma; P) \approx_{\mathrm{R}} (\emptyset, \mathbf{0})$.*

By extension, a (consistent) typed context $(\Sigma[\cdot], C[\cdot])$ is said inert *if up to structural congruence $C[\cdot]$ is of the form $(\boldsymbol{\nu}\tilde{a})\,(X \mid [\cdot])$ where X is inert.*

To prove discreetness of typable processes we are going to use a slightly stronger definition of bisimulation that only relates processes with the same type mapping, that allows working on simpler relations. Note that it is too strong as a general purpose bisimulation — for instance inert processes can't be defined with it.

Definition 4.8.2 (One-Type Discreet Bisimulation) *A relation $\mathcal{R}$ on typed processes is a* one-type discreet bisimulation *is it matches Definition 2.6.1 with the following changes:*

- *$(\Sigma_P; P)\mathcal{R}(\Sigma_Q; Q)$ implies $\Sigma_P = \Sigma_Q$.*
- *The third point is changed as: $\forall u : \sigma = \Sigma_P(u)$ and $\mathbf{md}(\sigma) = \uparrow_\omega$ implies $(\boldsymbol{\nu}u)\,(\mathrm{T}_\sigma(u) \odot \Sigma_P; \mathrm{L}_\sigma(u) \,|\, P)\ \mathcal{R}\ (\boldsymbol{\nu}u)\,(\mathrm{T}_\sigma(u) \odot \Sigma_Q; \mathrm{L}_\sigma(u) \,|\, Q)$.*

A one-type discreet bisimulation is not necessarily a discreet bisimulation ; however any one-type discreet bisimulation is contained in a discreet bisimulation, and in particular the largest one-type discreet bisimulation is (strictly) contained in $\approx_{\mathrm{R}}$.

We are going to use the following proof method for proving discreet bisimilarity, which is inspired by [San00].

Definition 4.8.3 (Discreet Bisimulations Up To Inert Contexts)
A relation $\mathcal{R}$ on typed processes is a (one-type) discreet bisimulation up to inert contexts *if it matches the definition for (one-type) discreet bisimulation for which all instances but the first of the relation $\mathcal{R}$ are replaced by $\mathcal{R}'$, for which $X\mathcal{R}Y$ implies $C[X]\mathcal{R}'C'[Y]$ for any inert contexts $C[\cdot]$ and $C'[\cdot]$.*

Lemma 4.8.4 *$\mathcal{R}$ being some (one-type) discreet bisimulation up to inert contexts, $A\mathcal{R}B$ implies $A \approx_{\mathrm{R}} B$.*

Lemma 4.8.5 *The relation $\mathcal{R} = \{(\Sigma_A; P) \mapsto (\Sigma_A; P) : A \vdash_{\mathrm{p}} P\}$ is a one-type discreet bisimulation up to inert contexts.*

Proof As $\mathcal{R}$ only relates equal processes, most points of the bisimulation definition are trivially proven, except for the uniqueness and closeness requirements:

Let $(\Sigma_A; P) \xrightarrow{a(\tilde{x})} (\Sigma'; P')$ and $\mathbf{md}(\Sigma_A(a)) = \downarrow_\omega$.

Data observable at a is unique by Lemma 4.6.1.

Let $a(\tilde{y}).Q$ be the input that was instantiated by this transition. By the input rule, there is B with $B \vdash Q$. As this input is replicated (Convention 3.6.1), $\forall b \notin \tilde{y} : \Sigma_B(b) \notin \{\downarrow_1, \uparrow_1, \downarrow_\omega, \bigstar_1\}$.

So, using Lemma 4.7.1, before applying the restriction, $(\boldsymbol{\nu}\tilde{x})\,(B\{^{\tilde{x}}/_{\tilde{y}}\})$ is inert and (α-renaming $\tilde{x}$ if needed) there is an inert context $C[\cdot]$ s.t. $(\boldsymbol{\nu}\tilde{x})\,P' \equiv C[P]$.

□

As a corollary, all typable processes are discreet.

References

[GNR04] M. Gamboni, U. Nestmann and A. Ravara. What is TyCO, After All? Master's thesis, École Polytechnique Fédérale de Lausanne, 2004.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil93] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer and H. Schwichtenberg, eds, *Logic and Algebra of Specification, Proceedings of the International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94. Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, U. K., 1991.

[MPW92] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–77, 1992.

[San00] D. Sangiorgi. The Bisimulation Proof Method. *Journal*, 2000. Entry To Be Completed.

[SW01] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[YBH04] N. Yoshida, M. Berger and K. Honda. Strong normalisation in the π-calculus. *Inf. Comput.*, 191(2):145–202, 2004.