Electronic Notes in Theoretical Computer Science 71 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume71.html 30 pages

Processes and Games

Kohei Honda [1,2]

Department of Computer Science, Queen Mary,
University of London, London, E1 4NS, UK.

Abstract

A general theory of computing is important, if we wish to have a common mathematical footing based on which diverse scientific and engineering efforts in computing are uniformly understood and integrated. A quest for such a general theory may take different paths. As a case for one of the possible paths towards a general theory, this paper establishes a precise connection between a game-based model of sequential functions by Hyland and Ong on the one hand, and a typed version of the π-calculus on the other. This connection has been instrumental in our recent efforts to use the π-calculus as a basic mathematical tool for representing diverse classes of behaviours, even though the exact form of the correspondence has not been presented in a published form. By redeeming this correspondence we try to make explicit a convergence of ideas and structures between two distinct threads of Theoretical Computer Science. This convergence indicates a methodology for organising our understanding on computation and that methodology, we argue, suggests one of the promising paths to a general theory.

Key words: Game semantics, typed π-calculus

1 Introduction

Computing in the modern world is characterised by diversity and rapid change. Science of computing is no exception, where we have diverse activities and theories dealing with different subject matters and using different approaches. All the more so there is a value in seeking a general theory of computing which can, among others, offer a common mathematical footing on which we can unify and integrate these diverse scientific theories and engineering disciplines. Such a general theory is expected to act on science and engineering of computing as basic physical theories have acted on natural sciences and engineering, in spite of a different nature of computing from physical phenomena (on this difference we do not extend further here; except noting the complexity

[1] Partially supported by EPSRC grant GR/N/37633
[2] Email: kohei@dcs.qmul.ac.uk

of computing systems – in particular software – nowadays *practically* requires their general and precise treatment as needed for natural phenomena). From a scientific viewpoint, a general theory would help us organise scientific knowledge on computing and for promoting further research based on the obtained unified understanding. From an engineering viewpoint. without such a unifying theory, it is hard to envisage how we can integrate diverse methodologies for building and controlling computing systems. Such integration becomes essential when we need to control the behaviour of an application consisting of distributed components written in different programming languages which, as a whole, serve users' needs.

Thus a search for a general theory of computing is important, both from scientific and engineering viewpoints. And a quest to reach a general theory may take different paths and methodologies. For example we may introduce a general algebraic framework which capture different notions of computation, as in *Concurrent term rewriting* introduced by Meseguer [33] and *tile model of computation* by Montanari and others [12], recent results from both of these strands being discussed in the present proceedings; another model developed with a similar goal is Milner's theory of bigraphs [39]. As exemplified in the present proceedings and in a recent work by Jensen and Milner [26], such an algebraic framework can offer a basis of integrated, and often illuminating, understanding of existing systems and theories, as well as a broad platform for experimentation and development of new concepts, methodologies and engineering ideas.

A different approach to a general theory would start from distinct, but equivalent, presentations of a theory for a core, if restricted, class of computation, so that one can find the fundamental shape of such a theory. Each of these different presentations serves a specific scientific/engineering need: as a whole, they offer a firm basis for ramifications and applications. After we reach a satisfactory account of the theory, which may in particular include the equivalence of different presentations, one may extend it to a broader class of computation. Hoare [16] laid down a strong case for this approach, focussing on a theory of basic imperative programming, for which he studies different (observational, algebraic and operational) ways of presenting the same theory, and showing they are equivalent to each other in the sense that each is derivable from another in a cyclic way. Different presentations not only serve different engineering purposes but also clarify distinct forms mathematical theories of computing can take: their mutual derivability allows flexibility in applications and adds the confidence to their mathematical status.

Finally, yet another approach would single out a concrete mathematical structure which can represent a large class of computational phenomena, and would try to develop a general theory on the basis of the structure. When the chosen structure has a wide repertoire for representing computation, and when this representation helps us reason about, and control, computational phenomena effectively (in comparison with directly dealing with them), this

method would particularly be successful. Theory of domains and denotational semantics, initiated by Scott and Strachy [45], is a forerunner of this approach.

These different approaches compensate each other: without ambient general algebraic theories, the understanding of specific structures would be a limited one; but a general algebraic theory may as well become most effective when combined with a powerful concrete structure. Different presentations of a core theory may as well be extended to a general algebraic universe, gaining in both richness and applicability; and the former may suggest an alternative, useful presentation of a general algebraic universe which would help its deeper understanding. In fact, any general theory, however powerful, can give only one way of abstracting computational phenomena: other abstraction would always be possible, and the interplay among different mathematical abstractions will enrich our scientific understanding. Thus distinct directions may as well be pursued, while allowing mutual enriching interactions.

This paper tries to place a small piece of a technical result in this broad arena of conversations, establishing an equivalence between a semantic universe introduced by Hyland and Ong [24], [3] on the one hand, and a theory of typed processes [5] centring on Milner, Parrow and Walker's π-calculus [40] on the other. The equivalence between these two theories, initially observed in [5], as well as its connection to Linear Logic [14] has been instrumental in our recent efforts to use the typed π-calculus as a basic tool for representing and analysing diverse classes of computations [5,6,47,23], even though the exact shape of correspondence has not been presented in a published form. By presenting this correspondence in a precise form, we hope to show how two distinct threads of studies have reached a common structure for representing computation. These two threads have quite different origins: the π-calculus is a calculus which, based on the preceding study on process algebra [4,15,35], tries to capture concurrent computation with dynamic change of structure in a simple syntax; while the semantic universe by Hyland and Ong, based on the preceding semantic studies on sequentiality (cf. [44,34,27,7,8,42]), was introduced for giving a precise semantic account of sequentiality in higher-order functions. Furthermore, the version of the π-calculus which is at the centre of this coincidence, is the minimal system which Boudol, Tokoro and I introduced in the beginning of 90's as a formalism for capturing distributed computation, whose dynamics is based on the following communication rule:

$$x(\vec{y}).P|\overline{x}\langle\vec{v}\rangle \longrightarrow P\{\vec{v}/\vec{y}\}.$$

The asynchronous communication is the unconstrained, fluid[35] form in comparison with the synchronous communication, maximising potential concurrent activities. Processes in this asynchronous version of the π-calculus are precisely those constituting interactions in the semantic universe of sequen-

[3] Closely related semantic universes were independently introduced by Abramsky, Jagadeesan and Malacaria [1] and by Nickau [41].

tiality, combined with a notion of types which impose a behavioural constraint by which processes behave just like sequential higher-order functions do.

The significance of this coincidence arises in two ways. First, it suggests the breadth of computational phenomena which are precisely representable by the operational structure found in both strands, name passing processes. [4] It may be safe to say that sequential pure functions are what the π-calculus was least expected to precisely capture: thus its representability in the π-calculus, obtained through its connection to Hyland-Ong games, strongly suggests similar results for other classes of computation, including diverse forms of concurrent and distributed computation. In other words, the precise representability of sequential higher-order functions in Hyland-Ong games gets positioned in a broader context by moving into the π-calculus.

Secondly, the technical result which relates Hyland-Ong games to the π-calculus accompanies a concrete method by which we may obtain precision in representation of computation for name passing processes. In the π-calculus, there have been a series of studies (starting from Milner [37], cf. [43,29,17,18]) on a notion of types called *sorting*, which essentially constrain the usage of names in processes and thus their behaviour. It turned out that a clean and simple type structure arises from the π-calculus representation of Hyland-Ong games which can in fact be positioned as a ramification of the existing notions of types for the π-calculus processes. A couple points are notable regarding this type structure. First it has several novel features in comparison with existing process types, among others duality (in a form close to Linear Logic [14]). Second, this type structure was only implicitly present in Hyland-Ong games, at least in its entirety: there, dynamic behavioural specifications on interaction, including *innocence*, are used in conjunction with static type structure. By moving to the π-calculus, a simple static type discipline is made explicit which in fact turns out to be equivalent to the original specifications. Third, and most interestingly, the resulting typed universe of processes articulates a broader class of sequential pure functions than in the original Hyland-Ong games, capturing, for example, call-by-value behaviour (see [5]; in Section 6 later we show how we can "read off" the universe of Hyland-Ong games from that of typed processes as a proper subset of the latter). By subsequent studies on game semantics and on typed π-calculi, both discussed below, we find the sequential types born from Hyland-Ong games constitutes one member of a uniform family of types for processes which characterise various forms of computation.

Thus the connection between the π-calculus and Hyland-Ong games not only indicates the significance of a certain operational structure they have in common; it also indicates a concrete way to *use* this structure for precise representability, suggesting a new tool and framework for studying computa-

[4] Regarding this point, Milner's address at Bologna [38] described the π-calculus as a calculus whose goal is the "analysis ... of informatic systems", comparing it to the differential calculus whose goal is the "analysis of physical systems".

tion. The π-calculus and its types based on duality together offer a uniform basis for precisely modelling the behaviour and algebras of different classes of computation, anticipating one possible form of a general theory.

We conclude this introduction with further discussions on the relationship of the present work to other studies. As we already mentioned, the π-calculus is one of the recent forms of process algebras. Different forms of process algebras (including ACP [4], CSP[15] and CCS/SCCS [35]) are based on different notions of synchronisations and process composition. The use of name passing in the π-calculus — its distinguishing feature — is better positioned as a way to enrich structures of interaction in a way orthogonal to distinction among different process calculi. In this sense, that a very simple form of name passing suffices to induce a large universe of computation is encouraging to explore larger universes in which different notions of synchronisations and other concurrency primitives are combined with name passing. It is notable that the common strata of many process algebras, among others parallel composition and hiding, play the essential rôle in the present theory: while its use of types and asynchrony in communication may constitute novel elements from the viewpoint of standard process algebras.

Connection between Hyland-Ong games and the π-calculus was observed by Hyland and Ong themselves [25]. The difference between the result in [25] and this work, as well as [5], is that [25] specifies processes based on behavioural characterisation which directly comes from their game semantics, while here and in [5] sequential processes are generated purely by a syntactic type discipline, from which behavioural characterisation is derived.

Syntactic and semantic theories of types have been studied in the context of sequential programming languages for decades, centring on, among others, the λ-calculus. The main difference between types for interaction and those for functions lies in the class of representable behaviour. For example, even the untyped λ-calculus arises as strongly typed processes, as we shall see in Section 3.6. Types for higher-order functions are a rich realm, with deep theories and powerful applications. One significant aspect of the connection between Hyland-Ong games and the typed π-calculus is its suggestion to the way by which types for processes may inherit the richness of types for functions.

As we already mentioned, the notion of types in the present work can be positioned among the variety of types for the π-calculus studied in the past, cf. [43,29,17,18]. In this context, the presented type discipline arises as a specific form of linear typing in which duality of types plays a fundamental role. Exploration of the precise positioning of the presented notion of types among a general universe of types for the π-calculus (as discussed in, for example, [19]) would be an interesting subject of study.

Abramsky, McCusker, Laird and others, as well as the present author, have explored different variants of Hyland-Ong games and have established embeddability of different language constructs in games, cf. [2,3,11,28,22,30,32]. These studies contribute to the identification of distinct universes of compu-

tation based on types for interaction, offering in-depth algebraic properties of each universe using the languages of logics and categories. As the present study reveals, the characterisations of typed interactions precisely correspond, and fundamentally differ in articulation, between games and the π-calculus. By positioning different notions of games in the context of the π-calculus, diverse operators and their algebras in games are recaptured on a common, and more terse, footing of name passing processes and their algebra; while logical and categorical articulation given by games can be an effective means for clarifying structures of typed interaction. A closely related field is the polarised versions of Linear Logic studied in, e.g. [31], which offer yet another, and this time proof-theoretic, articulations of classes of typed interactions studied by games and typed π-calculi (precise connection in this regard will be reported elsewhere). The interplay between these three strands of studies enrich our understanding on the common structure these studies are working at from different angles and using different technical tools, leading to its thorough and deeper understanding.

Finally we compare the sequential type discipline in the present work to its original version [5], the latter being used as a basis of our subsequent studies. The only difference in the two type disciplines lies in different ways in representing "choices". [5] uses *branching/selection types* (written $[\&_{i\in I}\vec{\tau}_i]^{\downarrow}$ and $[\oplus_i\vec{\tau}_i]^{\uparrow}$) for representing choices, which involve enriched dynamics similar to the sums in the λ-calculus and the additives in Linear Logic; while, in the present note, we solely use unary interaction for representing choices, which in fact corresponds to Hyland-Ong games (and also to [9,10], whose emphasis on untyped dynamics brings the idea closer to the choice in the present study). The presentation of choices used in [5] has the merit in that it offers a tractable syntax for representing various notions of choices, including general value passing and objects. On the other hand, the representation studied in the present work offers a more analytical view on choices: in fact, its operational structure directly corresponds to the standard protocol for encoding choices in the unary π-calculus, known since Milner's early work [36] (revealing yet another coincidence in these two threads of study). As a merit of a different kind, the syntactic type discipline for this protocol would suggest a possible way to type similar operational structures. Because of these interests, presenting an alternative form of type structure for choices, a fundamental notion of computing, may have a merit in its own right, apart from its correspondence with Hyland-Ong universe.

Structure of the Paper. In the remainder, Section 2 informally illustrates the basic ideas of the protocol of choices in the π-calculus, after introducing the latter's syntax. Section 3 introduces the syntax of the typed π-calculus with sums. Section 4 gives a short presentation of Hyland and Ong's universe of sequential higher-order recursion. Section 5 constructs a categorical universe from typed processes and show the equivalence between the resulting universe and the Hyland-Ong's universe.

Acknowledgement. My view on typed π-calculi as a core tool for a basic theory of computing has been enriched by a series of my ongoing collaborative work with Berger and Yoshida, starting from [5]. My warm thanks go to Ugo Montanari who invited me to the workshop on rewriting logic at Pisa, 2002, a lively discussion at which eventually led me to the writing of the present note. I particularly appreciate discussions with Jose Meseguer, both on technical and non-technical topics. I thank Fabio Gadduci for enjoyable discussions in the workshop as well as for his patience and relaxed attitude as an editor of the proceedings. This work is partially supported by EPSRC grant GR/N/37633.

2 Sums in the π-Calculus: a Preview

2.1 Protocol for Choices

Before going into technical discussions, we informally outline central ideas of operational structure and types for representing choices. The following is the grammar of the π-calculus which uses bound, asynchronous output. Let $x, y, \ldots$ and $a, b, \ldots$ range over an infinite collection of *names* (also called *channels* or *ports*).

$$P \ ::= \ x(\vec{y}).P \ \mid \ !x(\vec{y}).P \ \mid \ \overline{x}(\vec{y})P \ \mid \ P|Q \ \mid \ (\boldsymbol{\nu}\, x)P \ \mid \ \mathbf{0}.$$

$x(\vec{y}).P$ is an input, which receives a vector of names (to be instantiated in formal parameters $\vec{y}$ in P) via x. $!x(\vec{y}).P$ is its replicated version. $x(\vec{y}).P$ (resp. $!x(\vec{y}).P$) is often called *linear input* (resp. *replicated input*). An output $\overline{x}(\vec{y})P$ outputs a vector of new names $\vec{y}$. $P|Q$ is a parallel composition. We sometimes write $\Pi_{i\in I}P_i$ for the n-ary parallel composition of $\{P_i\}_{i\in I}$ (where if $I = \emptyset$ then $\Pi_{i\in I}P_i = \mathbf{0}$). $(\boldsymbol{\nu}\, x)P$ says x is local to P. $\mathbf{0}$ is the inaction, denoting the lack of behaviour. We assume $|$ is the weakest in precedence, so that $x(y).P|Q$ (resp. $\overline{x}(y)P|Q$, resp. $(\boldsymbol{\nu}\, x)P|Q$) denotes $(x(y).P)|Q$ (resp. $(\overline{x}(y)P)|Q$, resp. $((\boldsymbol{\nu}\, x)P)|Q$). The output prefix should be regarded as an asynchronous output in the sense that $\overline{x}(\vec{y})P$ corresponds to $(\boldsymbol{\nu}\, \vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ in the standard syntax. The structural congruence $\equiv$ is standard except it includes the rules for output asynchrony just mentioned, and is listed in Appendix. On processes modulo $\equiv$, the reduction rules are given as follows.

$$x(\vec{y}).P|\overline{x}(\vec{y})Q \longrightarrow (\boldsymbol{\nu}\, \vec{y})(P|Q)$$
$$!x(\vec{y}).P|\overline{x}(\vec{y})Q \longrightarrow !x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q)$$

The relation $\longrightarrow$ is closed under $|$, $(\boldsymbol{\nu}\, x)$ and $\overline{x}(\vec{y})$, but not under (linear and replicated) input prefixes.

Using this syntax, we outline how the choice is realisable by a series of

unary name passing, which follows [36,21].

$$\overline{x}(z_1 z_2)(z_1.P_1 | z_2.P_2) \mid x(z_1 z_2)\overline{z_1}$$

The process on the left-hand side, which offers two choices, sends to the continuation c two names, z_1 and z_2; then it waits with these names. If z_1 is invoked, then P_1 becomes active: similarly for z_2 and P_2. On the other hand, the process on the left-hand side, which selects information, receives two new names after the initial invocation, then selects the first one by outputting to it. As a result we obtain the following reduction:

$$\begin{aligned}\overline{x}(z_1 z_2)(z_1.P_1 | z_2.P_2) | x(z_1 z_2)\overline{z_1} &\longrightarrow (\boldsymbol{\nu}\, z_1 z_2)(z_1.P_1 | z_2.P_2 | \overline{z_1}) \\ &\longrightarrow P_1 | (\boldsymbol{\nu}\, z_2) z_2.P_2\end{aligned}$$

Above we assume neither z_1 nor z_2 occur in P_1 and P_2. Note $(\boldsymbol{\nu}\, z_2)z_2.P_2$ behaves as $\mathbf{0}$ since it can neither reduce by itself or interact with the outside. Thus P_1 is selected, instead of P_2. Note that, for this protocol to represent choice, it is important that the selecting side only invokes either z_1 or z_2, but not both: this is what the side offering the choice expects as the behaviour of the choosing side. On the other hand, the choosing side expects that there is exactly one occurrence of each of these names, so that selection is done in a deterministic way (one may also represent a nondeterministic choice by ramification, but in this paper we only consider this simple form, since it would give the basis of other related protocols). This "expectation" is an essential part of well-organised — or typed — behaviour of name passing processes, as we shall discuss below.

2.2 Typing Choices

Let us focus on the following subterm of the above process.

$$(z_1.P_1 | z_2.P_2)$$

For this process, the assumption on name usage mentioned above says that only one of z_1 and z_2 will be invoked, which is where the choice comes in. This means, for this term, the following reduction is natural:

$$(z_1.P_1 | z_2.P_2) \mid \overline{z}_1 \longrightarrow P_1$$

which is of course *not* correct from the viewpoint of *untyped processes*, since z_2 is still available for invocation: but, in *typed processes*. because we expect that the environment obeys the choice protocol, z_2 will never be invoked, hence $z_2.P_2$ will be safely garbage collected, justifying the reduction. Since this justification depends on the typing of the process, we may add an annotation as, for example: $(z_1.P_1 \,\&\, z_2.P_2)$, which, as an untyped process, is still the same

thing as $(z_1.P_1|z_2.P_2)$. Now we can write the reduction above as:

$$(z_1.P_1 \,\&\, z_2.P_2) \mid \overline{z}_1 \;\longrightarrow\; P_1.$$

The syntax is similar to the standard (guarded) sum, usually written $z_1.P_1 + z_2.P_2$, even though the present sum notation involves type information as its essential element. Thus, the underlying untyped process is in fact the summation-less π-calculus, combined by the standard parallel composition operator $|$. The involved dynamics can be justified via a basic untyped equational law as far as the underlying processes are well-typed.

This annotation, replacing some of $|$ with $\&$, is also essential for tractable syntactic discpline for choice, where we wish to type two possible choices, at z_1 and z_2, as a single collection. For example, we may type, assuming P_1 and P_2 has the same typing Γ:

$$\vdash (z_1.P_1 \,\&\, z_2.P_2) \triangleright z_1 : ()^{\downarrow} \& z_2 : ()^{\downarrow},\ \Gamma.$$

The type says it assumes that the environment would select either z_1 or z_2, and not both (here $\downarrow$ indicates a linear input: thus $()^{\downarrow}$ says a one-time input which does not carry any value). Dually we have a typing for selection:

$$\vdash \overline{z}_1 \triangleright z_1 : ()^{\uparrow} \oplus z_2 : ()^{\uparrow}.$$

Reading $\uparrow$ as an output, the typing this time indicates that it assumes the environment is waiting with two options, one at z_1 and another at z_2, and that there is a potential for the process to choose either z_1 or z_2. In this case, the process selects z_1. Naturally there is another "inhabitant" of this type, which is:

$$\vdash \overline{z}_1 \triangleright z_1 : ()^{\uparrow} \oplus z_2 : ()^{\uparrow},$$

selecting the left-branch.

As suggested by the notations, the input choice and the output choice have a natural notion of duality: in the above examples, the typings are strictly dual between input and output, at z_1 and z_2, both individually ($()^{\downarrow}$ and $()^{\uparrow}$) and collectively ($\&$ and $\oplus$). In composition, these dual typings annihilate each other, so that we obtain:

$$\vdash (z_1.P_1 \,\&\, z_2.P_2) \mid \overline{z}_1 \triangleright z_1 : \updownarrow,\ z_2 : \updownarrow,\ \Gamma$$

where $\updownarrow$ means no further composition is possible at the channel, which makes sense since there is no longer the possibility of the choice at either names. This duality is fundamental for having a coherent notion of composition: in fact, while we can type the output $\overline{z}_1$ as $\vdash \overline{z}_1 \triangleright z_1 : ()^{\uparrow}$, this process is not composable with the above input, since it does not accompany the other choice in the typing. This duality is fundamental for having a coherent universe of typed processes.

3 A Typed π-Calculus with Sums

3.1 Processes

In this section we formally introduce the typed π-calculus with sums which is the centre of the present study. As the syntax of processes, we use the same grammar as we gave in Section 2, except that the linear input "$x(\vec{y}).P$" is now replaced by "$\&_{i\in I}x_i(\vec{y}_i).P_i$" (with I being a finite set and $x_i \neq x_j$ for $i \neq j$). The process "$\&_i x_i(\vec{y}_i).P_i$" is called *the sum of* $\{x_i(\vec{y}_i).P_i\}$, which, as we already discussed, is best understood as a short-hand for the parallel composition $\Pi_{i\in I}x_i(\vec{y}_i).P_i$ under the assumption — later concretised as types — that the environment obeys the sum protocol. We may further add type annotation on bound names, which we omit for simpler presentation. The reduction for the linear input is accordingly extended:

$$\&_{i\in I}x_i(\vec{y}_i).P_i \mid \overline{x}_i(\vec{y}_i)Q \longrightarrow (\boldsymbol{\nu}\,\vec{y}_i)(P_i|Q)$$

which throws away the unchosen branches. It is important to remember that the reduction of sums given above only makes sense in the typed setting: however, when processes are indeed typed, we can recover the untyped dynamics from the typed dynamics, as we shall formally demonstrate later.

3.2 Channel Types

Channel types represent the structure of interaction a process would have at its channels. It uses four *action modes*, $\downarrow$, $\uparrow$, **!** and **?**. As the symbols indicate, $\downarrow$ and $\uparrow$ are mutually dual, while **!** and **?** are mutually dual. Channel types are then given by the following grammar.

$$\tau ::= \tau_{\mathtt{I}} \mid \tau_{\mathtt{O}} \mid \updownarrow \qquad \tau_{\mathtt{I}} ::= (\vec{\tau}_{\mathtt{O}})^{\downarrow} \mid (\vec{\tau}_{\mathtt{O}})^{\mathbf{!}} \qquad \tau_{\mathtt{O}} ::= (\vec{\tau}_{\mathtt{I}})^{\uparrow} \mid (\vec{\tau}_{\mathtt{I}})^{\mathbf{?}}$$

Here $\vec{\tau}$ indicates a vector of types. We call $\tau_{\mathtt{I}}$ *input type* and $\tau_{\mathtt{O}}$ *output type*. In each input/output type, an element of the vector inside the parenthesis of the type is *carried* in that type. We sometimes say a type of the form $(\vec{\tau})^p$ is a *p-type*. If τ is a p-type, we sometimes write τ^p. Given an input/output type τ, the *dual of* τ, written $\overline{\tau}$, is given by dualising action modes in τ inductively. We set $\mathsf{md}(\tau)$ as the outermost mode except $\mathsf{md}(\updownarrow) \stackrel{\text{def}}{=} \updownarrow$.

As may be guessed, an input type represents an input behaviour, similarly for an output type. A pair type indicates, at one channel, both input and output are present. We only use channel types which obey the following constraint:

- In $(\vec{\tau}_{\mathtt{O}})^{\downarrow}$, each carried type is **?**-type, dually for $(\vec{\tau}_{\mathtt{I}})^{\uparrow}$.
- In $(\vec{\tau}_{\mathtt{O}})^{\mathbf{!}}$, each carried type is either a **?**-type or a $\uparrow$-type, dually for $(\vec{\tau}_{\mathtt{O}})^{\downarrow}$.

In [5], essentially the same constraint was used, though a replicated type in [5] can carry only a unique linear type. In contrast, here a replicated type

can carry multiple linear types. As we shall see later this indicates the choice behaviour.

We define a partial commutative operation $\tau \odot \tau'$ by: (1) $\tau \odot \overline{\tau} = *$ ($\mathsf{md}(\tau) = \downarrow$) (2) $\tau \odot \overline{\tau} = \tau$ ($\mathsf{md}(\tau) = \mathbf{!}$) and (3) $\tau \odot \tau = \tau$ ($\mathsf{md}(\tau) = \mathbf{?}$). $\tau \odot \tau'$ is not defined if none of these rules apply, in which case we write $\tau \asymp \tau'$.

3.3 Action Types

We first define a *prime action type* as follows.

(1) $\&_{i\in I} x_i : \tau_i$ is a prime action type when $x_i \neq x_j$ if $i \neq j$ and $\mathsf{md}(\tau_i) = \downarrow$. Dually for $\oplus_{i\in I} x_i : \tau_i$. $\{x_i\}$ is called the *domain* of this prime action type.

(2) $x : \tau$ is a prime action type when $\mathsf{md}(\tau) \in \{\mathbf{!}, \mathbf{?}\}$. $\{x\}$ is called the *domain* of this prime action type.

A prime action type of the form (1) is sometimes called *sum type*. Then an *action type* is a finite collection of prime action types such that any two of their domains are disjoint from each other. [5] $\Gamma, \Delta, \ldots$ range over action types. We often regard an action type as the underlying finite map from names to action types, writing $\Gamma(x)$ for the image of Γ at x and $\mathsf{dom}(\Gamma)$ for the domain of, or the named used in, Γ. Note Γ is determined by the underlying map together with the groupings by & and $\oplus$ of linearly typed names. We also write $\Gamma/\vec{x}$ for the result of taking off $\vec{x}$ from the domain of Γ.

Next we define the partial algebra $\odot$ on action types.

Definition 3.1 $\Gamma \asymp \Delta$ when the following conditions are all satisfied:

(i) For each $x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)$, $\Gamma(x) \asymp \Delta(x)$.

(ii) If $\&_i x_i : \tau_i \in \Gamma$ and $x_i \in \mathsf{dom}(\Delta)$ then $\oplus_i x_i : \overline{\tau}_i \in \Delta$; dually if $\oplus_i x_i : \tau_i \in \Gamma$ and $x_i \in \mathsf{dom}(\Delta)$ then $\&_i x_i : \overline{\tau}_i \in \Delta$.

(iii) The symmetric case of (ii).

If $\Gamma \asymp \Delta$, we set $\Gamma \odot \Delta$ as follows: $(\Gamma \odot \Delta)(x) = \tau$ iff either (1) $x \notin \mathsf{dom}(\Delta)$ and $\tau = \Gamma(x)$, (2) $x \notin \mathsf{dom}(\Gamma)$ and $\tau = \Delta(x)$, or (3) $x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)$ and $\tau = \Gamma(x) \odot \Delta(x)$. Further a sum type is in $\Gamma \odot \Delta$ iff either it is in $\Gamma/\mathsf{fn}(\Delta)$ or in $\Delta/\mathsf{fn}(\Gamma)$.

By the algebra, when we compose Γ and Δ, the resulting type can only contain a prime action type which comes from either Γ or Δ, or the result of composing prime action types at common channels. Note a sum prime action type is treated as a whole, involving multiple names in general. This constraint is essential for the consistency of the type discipline.

[5] In many examples (including types representing games in Hyland-Ong games), an action type does not contain two $\oplus$-prime action types, or two &-prime action types. In such cases, it is not necessary to group them by $\oplus$ and &. However the groupings by prime action types become necessary in typed processes in general, for example in labelled transition.

3.4 Tying Rules

The sequent has the form $\vdash_\phi P \triangleright \Gamma$ where ϕ is an *IO-mode*, an element from the set $\{\mathtt{I}, \mathtt{O}\}$. We write $\phi \asymp \psi$ when either (or both) of ϕ and ψ is $\mathtt{I}$. If so $\phi \odot \psi$ is $\mathtt{I}$ if both are, and is $\mathtt{O}$ if else.

We introduce the typing rules one by one. In each rule, we assume channel/action types introduced in the antecedent are well-formed. The basic composition rules are straightforward, which are from [5].

$$\frac{}{\vdash_{\mathtt{I}} \mathbf{0} \triangleright \emptyset}\ (\mathsf{Zero}) \qquad \frac{\vdash_{\phi_i} P_i \triangleright \Gamma_i \quad (i=1,2) \quad \Gamma \asymp \Delta \quad \phi \asymp \psi}{\vdash_{\phi\odot\psi} P|Q \triangleright \Gamma \odot \Delta}\ (\mathsf{Par})$$

$$\frac{\vdash_\phi P \triangleright \Gamma, x{:}\tau \quad \mathsf{md}(\tau) \in \{\mathbf{!}, \updownarrow\}}{\vdash_\phi (\boldsymbol{\nu}\, x)P \triangleright \Gamma}\ (\mathsf{Res}) \qquad \frac{\vdash_\phi P \triangleright \Gamma^{\text{-}x} \quad \mathsf{md}(\tau) \in \{\mathbf{?}, \updownarrow\}}{\vdash_\phi P \triangleright \Gamma, x{:}\tau}\ (\mathsf{Weak})$$

In (Par) we use the partial algebra on types and IO-modes, so that the resulting process is always sequential. In (Res) we do not hide a channel which indicates the need for its dual — that which has either $\downarrow$, $\uparrow$ or $\mathbf{?}$-mode. In (Weak), $\Gamma^{\text{-}x}$ indicates $x \notin \mathsf{dom}(\Gamma)$. The weakening intuitively says that it is OK not to use $\mathbf{?}$-channels at all; and that a $\updownarrow$-channel can be absent since linear input and output annihilate each other.

Next we introduce the linear prefix rules, which are the only place we introduce the idea of choices.

$$\frac{\vdash_{\mathtt{O}} P_i \triangleright \uparrow \mathbf{?}\Gamma^{\text{-}\vec{x}}, \vec{y_i}{:}\vec{\tau_i} \quad (1 \le i \le n)}{\vdash_{\mathtt{I}} \&_i x_i(\vec{y_i}).P_i \triangleright \Gamma,\ \&_i x_i{:}(\vec{\tau_i})^\downarrow}\ (\mathsf{In}^\downarrow) \qquad \frac{\vdash_{\mathtt{I}} P \triangleright \Gamma, \vec{y}{:}\vec{\tau_i} \quad \Gamma \asymp \oplus_i x_i{:}(\vec{\tau_i})^\uparrow}{\vdash_{\mathtt{O}} \overline{x}_i(\vec{y_i})P \triangleright \Gamma \odot \oplus_i x_i{:}(\vec{\tau_i})^\uparrow}\ (\mathsf{Out}^\uparrow)$$

In $(\mathsf{In}^\downarrow)$, $\Gamma^{\text{-}\vec{x}}$ is understood as before, indicating $\mathsf{dom}(\Gamma) \cap \{\vec{x}\} = \emptyset$. Some observations on these rules:

- In $(\mathsf{In}^\downarrow)$, $\Gamma^{\text{-}\vec{x}}$ ensures linearity of $x_1, .., x_n$. Here, as illustrated in Section 2, "$\&_i x_i{:}(\vec{\tau_i})^\downarrow$" indicates the provision of choices. Note also Γ only contains output types so that an input prefix does not suppress another input, and the output mode is turned into the input mode. This follows [5], and is closely related to so-called *input-output alternation* in game semantics [1,1].
- $(\mathsf{Out}^\uparrow)$ introduces a collection of selection types, dual to $(\mathsf{In}^\downarrow)$. "$\oplus_i x_i{:}(\vec{\tau_i})^\uparrow$" indicates possible selections a process may do. $(\mathsf{Out}^\uparrow)$ has the input-output alternation dual to $(\mathsf{In}^\downarrow)$, turning an input mode to an output mode. Finally the use of $\asymp$ and $\odot$ indicates the asynchronous character of output.

We conclude with the typing rules with replicated input and output.

$$\frac{\vdash_{\mathtt{O}} P \triangleright ?\Gamma^{-x}, \vec{y}{:}\vec{\tau}}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright \Gamma, x:(\vec{\tau})^{!}}\ (\mathsf{In}^{!}) \qquad \frac{\vdash_{\mathtt{I}} P \triangleright \Gamma, \vec{y}{:}\vec{\tau} \quad \Gamma \asymp x:(\vec{\tau})^{?}}{\vdash_{\mathtt{O}} \overline{x}(\vec{y})P \triangleright \Gamma \odot x:(\vec{\tau})^{?}}\ (\mathsf{Out}^{?})$$

Neither of these rules involves choices: this is because these names can be used as many times as needed. In $(\mathsf{In}^{!})$, the suppressed channels in Γ cannot contain a linear output since, by being under a replication, these channels can be used for outputs for arbitrarily many times. $(\mathsf{Out}^{?})$ is the same as $(\mathsf{Out}^{\uparrow})$ except for the lack of sum typing. Simple examples of typed processes follow.

Example 3.1 (i) Let $\mathbf{t}\langle x\rangle \stackrel{\text{def}}{=} !x(b_1b_2).\overline{b_1}$ and $\mathbf{f}\langle x\rangle \stackrel{\text{def}}{=} !x(b_1b_2).\overline{b_1}$. Then, with R being one of $\mathbf{t}\langle x\rangle$ or $\mathbf{f}\langle x\rangle$, we have $\vdash_{\mathtt{I}} R \triangleright x:((\,)^{\uparrow}(\,)^{\uparrow})^{!}$. $\mathbf{t}\langle x\rangle$ and $\mathbf{f}\langle x\rangle$ represent (call-by-name) truth and falsity, respectively. Their dual is the conditional, given as: $\mathtt{if}\, x\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2 \stackrel{\text{def}}{=} \overline{x}(b_1b_2).(b_1.P_1 \,\&\, b_2.P_2)$. which is typed as $\vdash_{\mathtt{O}} \mathtt{if}\, x\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2 \triangleright x : ((\,)^{\downarrow}(\,)^{\downarrow})^{?} \otimes A$, assuming $\vdash_{\mathtt{O}} P_i \triangleright A$ $(i = 1, 2)$, We can then check $\mathbf{t}\langle x\rangle \mid \mathtt{if}\, x\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2 \longrightarrow^2 \mathbf{t}\langle x\rangle \mid P_1$ and, symmetrically, $\mathbf{f}\langle x\rangle \mid \mathtt{if}\, x\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2 \longrightarrow^2 \mathbf{f}\langle x\rangle \mid P_2$.

(ii) We can also construct the call-by-value versions of these agents. In this case the truth and false become $\vdash_{\mathtt{O}} \overline{b}_1 \triangleright b_1:(\,)^{\uparrow} \oplus b_2:(\,)^{\uparrow}$ and $\vdash_{\mathtt{O}} \overline{b}_2 \triangleright b_1:(\,)^{\uparrow} \oplus b_2:(\,)^{\uparrow}$, while the conditional becomes $\vdash_{\mathtt{I}} b_1.P_2 \,\&\, b_2.P_2 \triangleright b_1:(\,)^{\downarrow} \& b_2:(\,)^{\downarrow}$. An indirection in the call-by-name encoding is eliminated in these encodings.

3.5 Basic Syntactic Properties

We first list basic properties of the dynamics in sequential processes. Below we let $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup \longrightarrow^{*}$.

Proposition 3.2 (i) (subject reduction) *If* $\vdash_{\phi} P \triangleright \Gamma$ *and* $P \twoheadrightarrow Q$ *then* $\vdash_{\phi} Q \triangleright \Gamma$.

(ii) (sequentiality) *If* $\vdash_{\mathtt{O}} P \triangleright \Gamma$ *and* $P \longrightarrow P'_{1,2}$ *then* $P'_1 \equiv P'_2$,

Proof. See Appendix. □

Next we formally establish the relationship between the untyped calculus and the typed calculus in their dynamics. We first define $\mathsf{Erase}(\vdash_{\phi} P \triangleright \downarrow\uparrow \mathbf{!?}A, \updownarrow \vec{w})$ as $(\boldsymbol{\nu}\, \vec{w})P'$ where P' is the result of turning each & in P into parallel composition. Since names of mode $\updownarrow$ are never composed with other names, hiding them does not influence algebra and dynamics. Further write $\equiv'$ for the result of adding the axiom $(\boldsymbol{\nu}\, x)x(\vec{y}).P \equiv \mathbf{0}$ to $\equiv$ (this equation is justifiable by the untyped strong bisimilarity, so that it has no effect on the behaviour of processes). We can now state the following. The proof is easy by inspecting the two generation rules for reduction.

Proposition 3.3 *Let* $\vdash_\phi P \triangleright A$ *and* $P_0 \stackrel{def}{=} \mathsf{Erase}(\vdash_\phi P \triangleright A)$. *Then:*

(i) $P \longrightarrow P'$ *implies* $P_0 \longrightarrow \equiv' P_0'$ *where* $P_0' \stackrel{def}{=} \mathsf{Erase}(\vdash_\phi P' \triangleright A)$.

(ii) $P_0 \longrightarrow P_0'$ *implies* $P_0' \equiv' \mathsf{Erase}(\vdash_\phi P' \triangleright A)$ *such that* $P \longrightarrow P'$.

Finally we list one property which we shall repeatedly use in the next section. Let us say a process P is *prime with subject* x, or simply *prime*, if either P is input with subject x or $P \equiv \overline{x}(y_1..y_n)\Pi_{i\in I}P_i$ such that each P_i is prime with subject y_i. Then we consider a variant of the typing for output prefixes which is given by adding the condition "$P \equiv \Pi P_i$ with P_i prime with subject y_i" in the premise of $(\mathsf{Out}^{\uparrow,?})$. This restricted typing system is called *prime typing*. Note that, in the prime typing system, we can assume active names under an output prefix are bound by that prefix. We can then easily check:

Proposition 3.4 *If* $\vdash P \triangleright A$ *is derivable in the original typing rules then for some* $P_0 \equiv P$ *we have* $\vdash P_0 \triangleright A$ *in the prime typing system.*

Proposition 3.4 says that we can assume, without loss of generality, that all prefixed processes are primes whenever we are discussing properties invariant under $\equiv$.

3.6 Infinitary Extension

In this subsection we extend the π-calculus to its infinitary counterpart, both in types and processes. While this is not necessary, at least in its most general form, in order to capture many standard computational behaviours, the infinitary extension is needed for having the precise equivalence with the original category by Hyland and Ong [24]. It also has some interest in modelling untyped sequential calculi, as we shall see later. The construction is quite simple.

A process is now a possibly infinite tree of at most countable height [6] such that each of its full subtrees [7] has the following shape:

$$\&_{i\in I}x_i(\vec{y_i}).P \;\mid\; !x(\vec{y}).P \;\mid\; \overline{x}(\vec{y})P \;\mid\; \Pi_{i\in I}P_i \;\mid\; (\boldsymbol{\nu}\,\vec{x})P \;\mid\; \mathbf{0}.$$

with I and each vector of names being possibly infinite (for simplicity assume such a vector are indexed by an initial segment of ordinals, starting from 0). To make the α-conversion possible under infinite name occurrences, we set the whole collection of names to be a proper class. The resulting processes are sometimes called *infinitary*, which include the original finitary processes as a proper subset. On infinitary processes we define $\equiv$ using the obvious extension of the original equations (in particular we assume nested Π_i is commutative and associative with the identity $\mathbf{0}$; the rules involving hiding are extended to

[6] The restriction to countable height is not substantial, but makes many discussions simpler.

[7] A *full subtree* T of a tree T_0 is a subtree of T_0 such that, at each vertex, T contains all branches of those of T_0 at the corresponding vertex.

possibly infinite vectors, as in $(\vec{x})(\vec{y})P \equiv (\vec{y})(\vec{x})P$). The reduction rules are given precisely as before.

Channel types are similarly extended, allowing each vector to be finite or infinite and each type to be of a finite or countably infinite height, with the same well-formedness conditions as before. A sum type can have an infinite domain; so is an action type in general. Using infinitary processes and types, the typing rules are given precisely as before, reading each rule as a constraint on each full subtree of a process.

Example 3.2 (representation of untyped λ-calculus) The infinitary extension has some interest from the viewpoint of representation of untyped higher-order functions. The following is an encoding of the untyped λ-calculus, known to the present author since 1996.

$$\begin{array}{rll} [\![x]\!]_u \stackrel{\text{def}}{=} \mathsf{CC}\langle ux\rangle & & \\ [\![\lambda x.M]\!]_u \stackrel{\text{def}}{=} !u(x\cdot\{y_i\}_{i\in\omega}).P & & ([\![M]\!]_m \stackrel{\text{def}}{=} !u(\{y_i\}_{i\in\omega}).P) \\ [\![MN]\!]_u \stackrel{\text{def}}{=} !u(\{y_i\}_{i\in\omega}).(\boldsymbol{\nu}\, x)(P|[\![N]\!]_x) & & ([\![M]\!]_m \stackrel{\text{def}}{=} !u(x\cdot\{y_i\}_{i\in\omega}).P) \end{array}$$

where we set $\mathsf{CC} \stackrel{\text{def}}{=} \mu X\langle ab\rangle.!a(\{y_i\}_{i\in\omega}).\overline{b}(\{z_i\}_{i\in\omega})\Pi_i X\langle z_i y_i\rangle$, which defines an infinite tree as a least fixed point ($X\langle ab\rangle$ etc. indicates the instantiation by names in the obvious way). In the definitions above, $\{y_i\}_{i\in\omega}$ is a vector of names indexed by the set of natural numbers; $x\cdot\{y_i\}_{i\in\omega}$ inserts x at the initial position of the vector, and shifts the indices of the remaining ones by one (note this results in the vector with the same indexing set). Let us show how $(\lambda x.x)M \longrightarrow M$ is modelled. Below we omit the indexing set ω, and let $[\![M]\!]_x \stackrel{\text{def}}{=} !x(\{z_i\}).P$.

$$\begin{array}{rcl} [\![(\lambda x.x)M]\!]_u & \stackrel{\text{def}}{=} & !u(\{y_i\}).(\boldsymbol{\nu}\, x)(\overline{x}(\{z_i\})\Pi_i\mathsf{CC}\langle z_i y_i\rangle \mid [\![M]\!]_x) \\ & \longrightarrow & !u(\{y_i\}).(\boldsymbol{\nu}\, \{z_i\})(\Pi_i\mathsf{CC}\langle z_i y_i\rangle \mid P) \\ & \approx & !u(\{z_i\}).P \stackrel{\text{def}}{=} [\![M]\!]_u. \end{array}$$

In the first line, note $\lambda x.x \stackrel{\text{def}}{=} !u(x\cdot\{y_i\}).\overline{x}(\{z_i\})\Pi_i\mathsf{CC}\langle z_i y_i\rangle$. In the last line, $\approx$ is the standard (untyped) weak bisimilarity, whose establishment is easy by using a closure following [36]. In this way we can justify both β and η-equalities: in fact, the encoded processes modulo $\approx$ capture the untyped λ-calculus up to the standard maximal consistent theory. Being independently conceived, the encoding is closely related to the games models of the untyped λ-calculus in [13,28]. In comparison, the process encoding is terse in presentation and directly captures the dynamics of the original calculus; while the models in [13,28] offer rich algebraic insights on ambient semantic structures.

4 Hyland-Ong Games

4.1 Arenas

Hyland and Ong games were introduced as an intensional structure for representing sequential higher-order recursion (as noted in Introduction, Abramsky, Jagadeesan and Malacaria [1] and Nickau [41] introduced closely related, and essentially equivalent, semantic universes around the same time). In this section we give a concise presentation of their universe.

The construction by Hyland and Ong is based on two notions, *arenas* and *strategies*. We first define arenas. Below a *forest* is an acyclic directed graph $(X, \mapsto)$ (writing $x \to y$ if there is a directed edge from x to y for $x, y \in X$) in which there is at most one incoming edge for each node and for which there are minimal elements w.r.t. the partial order $\mapsto^*$. These minimal elements are called its *roots*. We only consider forests of at most countable height (having uncountably high forests does not pose any technical problem, but is insignificant). We count the *height* of each node in a forest, starting from 1 at each root.

Definition 4.1 An *arena*, ranged over by $A, B, \ldots$ is a (possibly infinite, but of at most countable depth) forest $(X, \mapsto)$ (with $x, y, \ldots \in X$) together with two labelling functions $\mathsf{op} : X \to \{O, P\}$ and $\mathsf{qa} : X \to \{Q, A\}$ such that:

(i) If x is of an odd-height (resp. even-height) then $\mathsf{op}(x) = O$ (resp. $\mathsf{op}(x) = P$). Further if x is a root then $\mathsf{qa}(x) = Q$.

(ii) If $\mathsf{qa}(x) = A$ then for no y we have $x \mapsto y$.

Nodes in an arena are often called *moves*.

The op-labelling is redundant: however having it explicitly is convenient for establishing its connection with the π-calculus. O stands for an *opponent* while P stands for a *player*. On the other hand, Q and A stands for *question* and *answer*. We set the dualisation operator as $\overline{O} = P$ and $\overline{P} = O$ (there is no dualisation on qa-labels). We combine these modes as OQ, OA, PQ and PA. The following shorthand notations for these actions are used in [24]: [stands for an OQ-move, (for a PQ-move,) for an OA-move, and] for a PA-move.

Example 4.1 A simple example of an arena is a *boolean arena* $\mathbb{B}$, which has the single root labelled as OQ and two subsequent nodes both labelled as PA (which may be written `true` and `false`), and no more. Later we shall see $\mathbb{B}$ corresponds to an action type $x : ((\,)^{\uparrow}(\,)^{\uparrow})^{!}$. In fact, if we extend its channel type $((\,)^{\uparrow}(\,)^{\uparrow})^{!}$ as a syntax tree, the root has the $!$-mode, which we regard as an OQ-, or [-, move; and each $(\,)^{\uparrow}$ can simply be considered as the node of label $\uparrow$, which we regard as a PA-, or]-, move, giving as a whole $\mathbb{B}$. We can guess, in this way, $!$, $?$, $\downarrow$ and $\uparrow$ respectively correspond to [, (,) and] (in particular, and perhaps confusingly, Opponent corresponds to input and Player corresponds to output).

4.2 Strategies

We next construct strategies. Our presentation benefits from [32]. Write $\overline{A}$ for the *dualisation of* A, which exchanges OP-labelling but not QA-labelling. Given an arena A and B, an arena $A \Rightarrow B$ is made by combining $\overline{A}$ and B disjointly and adding a directed edge from each root of B to each (original) root of $\overline{A}$. Note this is again an arena, called the *function arena from* A *to* B. Below *a (finite) sequence* is a mapping from a finite initial segment of ω, the set of natural numbers, to a set of elements. The elements of the associated initial segment are called its *indices*. If σ is a sequence and i is its index, then $\sigma[i]$ denotes the i-th element of σ.

Definition 4.2 An *action sequence from* A *to* B is a finite sequence σ of moves in $A \Rightarrow B$ together with a *justification relation* on its indices, written $i \curvearrowright j$ (where i, j are indices of σ), such that:

(i) (initial move) If $\sigma = x\sigma'$ then x is a root of $A \Rightarrow B$.

(ii) (justification) If $i \curvearrowright j$ then $\sigma[i] \mapsto \sigma[j]$ in $A \Rightarrow B$. Further if $i_1, i_2 \curvearrowright j$ then $i_1 = i_2$.

(iii) (IO-alternation) If $\sigma = \sigma_1 \cdot x \cdot y \cdot \sigma_2$ then $\mathsf{op}(x) = \overline{\mathsf{op}(y)}$.

Two action sequences are equal if they coincide both in the underlying sequences and their justification relations.

Fixing an action sequence σ, we sometimes write $x^i \curvearrowright y^j$, or even $x \curvearrowright y$ when there is no ambiguity, for $i \curvearrowright j$ such that $\sigma(i) = x$ and $\sigma(j) = y$. When $x^i \curvearrowright y^j$, then x^i is the *justifying move* of y^j. If $x^i \curvearrowright y^j$ and y is an answer then we say y^j *is answered by* x^i. We often write $\sigma \cdot x$ to denote a concatenated sequence together with an implicit justification to x from some move in σ.

Given σ, its *player view* $PV(\sigma)$ is given as:

$$\begin{aligned} PV(\varepsilon) &= \varepsilon \\ PV(\sigma \cdot x) &= PV(\sigma) \cdot x && \mathsf{op}(x) = P \\ PV(\sigma_1 \cdot x \cdot \sigma_2 \cdot y) &= PV(\sigma_1) \cdot x \cdot y && \mathsf{op}(x_n) = O, x \mapsto y \\ PV(\sigma \cdot x) &= x && x \text{ is a root} \end{aligned}$$

where, in the second line (resp. the third line), we assume the justification relation in σ as well as on x (resp. on x and y) are preserved in the obvious way: for example, in the third line, we assume $x \mapsto y$ again in the new sequence for the mentioned occurrences of x and y. Dually we define the opponent view $OV(\sigma)$. We can now define legal action sequences.

Definition 4.3 (i) (visibility) σ is *P-visible* if, for each prefix $\sigma' \cdot x$ of σ such that $\mathsf{op}(x) = P$, the justifying move of x in σ occurs in $PV(\sigma)$. σ is *O-visible* if the dual condition is satisfied. It is *visible* if it is both P-visible and O-visible.

(ii) (well-bracketing) σ is *well-bracketing* if no question is answered earlier than a later question.

(iii) (legal sequences) σ is *legal* if it is visible and well-bracketing.

Well-bracketing means, simply put, the parentheses match properly when written using the notations [, (,) and], taking the justification into consideration (e.g. [can only be closed by] which it justifies). We can now define an innocent strategy.

Definition 4.4 An *innocent strategy f from A to B* is a prefix-closed set of action sequences from A to B satisfying the following condition:

(i) (contingency completeness) Whenever $\sigma \in f$ and $\sigma{\cdot}x$ is legal with $\mathsf{op}(x) = O$, we have $\sigma \cdot x \in f$.

(ii) (innocence) Whenever $\sigma_1, \sigma_2 \in f$ where both end with an opponent move and $PV(\sigma_1) = PV(\sigma_2)$, $\sigma_1{\cdot}x \in f$ implies $\sigma_2{\cdot}x \in f$ such that $PV(\sigma_1{\cdot}x) = PV(\sigma_2 \cdot x)$.

We write $f : A \to B$ when f is an innocent strategy from A to B.

By innocence and contingency completeness, an innocent strategy is precisely characterised by a partial function from odd-length legal player views to next actions (if any). Such functions are called *innocent functions*. An innocent function uniquely defines an innocent strategy and vice versa. In essence, contingency completeness says that a strategy is always ready to receive any legal input; innocence says that a strategy always reacts in the same way in the same context (where the sameness in both instances takes the notion of justification on sequences into account).

Example 4.2 Let $\mathbf{1}$ be the empty arena. An innocent strategy from $\mathbf{1}$ to $\mathbb{B}$ is that which returns `true` after the initial move (the latter justifies the former); another returns `false`; and the third one returns nothing. These respectively correspond to the three constants, the truth, the falsity, and the undefined, which are all and the only inhabitants of the boolean type. An innocent function from $\mathbb{B}$ to $\mathbb{B}$ starts from the root of the co-domain, and returns `true` immediately at the co-domain. This is a constant function of value `true`. Another innocent function of the same type would, after the initial action at the co-domain, asks at the domain, receives `true` (resp. `false`) then outputs `false` (resp. `true`) at the co-domain, defining the negation.

We may visualise an action sequence in a strategy using the two rows corresponding to its co-domain and domain. For example, writing $\bullet$ for the unique initial move, we may draw the following picture for representing the longest action sequence in the "truth" strategy:

$$\begin{array}{lll} \overline{\mathbf{1}} & : & \\ \mathbb{B} & : & \bullet \quad \texttt{true} \end{array}$$

We should further draw the justification, which leaves implicit. Similarly we can write one of the two longest sequences in the negation strategy as follows (another is its symmetric case):

$\overline{\mathbb{B}}$	:		•	`true`
$\mathbb{B}$	:	•		`false`

We again omit justification. In the next subsection we shall see how the above two strategies can be composed to induce the falsity.

4.3 Composition and Category of Games

Proposition 4.5 *An action sequence is* well-knit *if a root occurs only as its initial move. We write* $\mathsf{wk}(f)$ *for the subset of sequences which are well-knit. Then* $\mathsf{wk}(f) = \mathsf{wk}(g)$ *iff* $f = g$ *for any* $f, g : A \to B$.

Proof. Since if $\mathsf{wk}(f) = \mathsf{wk}(g)$ the actions of f and g after the identical P-views coincide by the construction of P-views. □

We note Hyland-Ong defined their innocent strategies solely in terms of well-knit sequences. We use non-well-knit sequences since this form is more convenient for composition of strategies.

The composition of two strategies can be defined in various ways: here we use the 4-row based presentation (cf. [32,22]). Let $f : A \to B$ and $g : B \to C$. A *composite sequence* from f and g is an array of the following shape (with justification pointers implicit), such that:

(i) In the $\overline{A}$-B rows, we write an action sequence from f. In the $\overline{B}$-C rows, we write a well-knit action sequence from g.

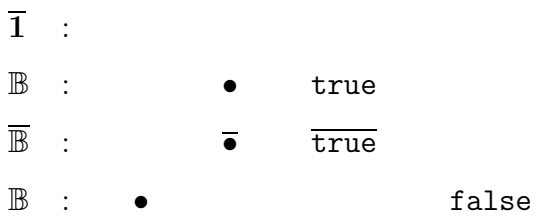
(ii) The initial entry (if any) is in the C-row.

(iii) The entries at B and $\overline{B}$ should always be both empty, or both non-empty and coincide as moves (with OP-labelling dualised).

$\overline{A}$	:				$\overline{z}$			...
B	:		y			w		...
$\overline{B}$	:		$\overline{y}$			$\overline{w}$		...
C	:	x					u	...

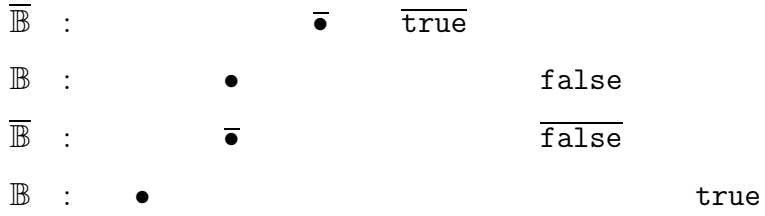
Above we write $\overline{y}$ etc. to show the dualisation w.r.t. OP-labelling. In these four rows, we call the first and fourth rows ($\overline{A}$ and C above) *visible rows*. Let us see how we can compose action sequences using this idea by examples.

Example 4.3 (i) We first compose the truth strategy and the negation:



Here the projection onto the visible rows give the behaviour of the falsity (as should be expected): the strategy receives a question at $\mathbb{B}$, then it in effect answers by the (justified) `false`.

(ii) Next we consider the negation composed with itself.



If we focus on the visible rows, it first receives an opponent question at $\mathbb{B}$, to which it reacts (in effect) by asking the opponent at $\overline{\mathbb{B}}$. If it receives `true` as a result, then it (in effect) reacts at the original type by answering by `true`, i.e. precisely by the same value as it has received from the opponent. The composite sequence when the opponent answers by `false` at the third step is precisely symmetric. These composite sequences suggest this composition results in the identity on $\mathbb{B}$.

Given $f : A \to B$ and $g : B \to C$ and a composite sequence from f and g, we can check the projection of the latter to the visible rows give an action sequence from A to C by adding the justification from the root moves in C to those in A, which we call *the visible projection* of the composite sequence. The set of the visible projection of all composite sequences from $f : A \to B$ and $g : B \to C$ is called the *composition of f and g*, which we write $f; g$. Hyland and Ong [24] showed that, for each $f : A \to B$ and $g : B \to C$, $f; g$ always gives an innocent strategy from A to C. Further they showed:

Proposition 4.6 *The following data defines a Cartesian-closed category, which we write* $\mathbb{CA}$.

(i) *Objects: arenas.*

(ii) *Arrows: innocent strategies.*

(iii) *Composition of arrows: the composition of innocent strategies.*

In the next section we construct $\mathbb{CA}$ using processes in the typed π-calculus.

5 Equivalence

5.1 Sequential Transition

This section constructs a category from typed processes in Section 3 and show that it coincides with $\mathbb{CA}$ in Section 4. For this purpose we select action types of a specific form, and take them modulo a typed weak bisimilarity, which become morphisms in a category, which in fact coincides with $\mathbb{CA}$. The bisimilarity is defined by the typed labelled transition which makes explicit the sequential behaviour of a typed process when interacting with another typed process. The rules generate transition for infinitary processes, though they do not differ those for finitary processes. We use the following labels for transitions.

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

where vectors can be infinitary. The transition is defined on terms modulo $\equiv$, so that, via Proposition 3.4, we can safely assume all processes are typed under the alternative typing. The transition rules follow [5], with an additional treatment of sums.

We start with the linear input. In this rule as well as the remaining ones, we assume the process on the left-hand side in the conclusion is well-typed (which, as we shall see later, implies the same for the process on the right-hand side.)

$$(\mathsf{In}^{\downarrow}) \ \frac{-}{\vdash_{\mathtt{I}} \&x_i(\vec{y}_i).P_i \triangleright \Gamma, \&_i x_i : (\vec{\tau}_i)^{\downarrow} \quad \stackrel{x_i(\vec{y}_i)}{\longrightarrow} \quad \vdash_{\mathtt{O}} P_i \triangleright \Gamma, \vec{y}_i : \vec{\tau}_i} \ \Delta =$$

The rule says that the i-th branch of the branching input is selected by interaction with the environment, and, as the result, all other branches are discarded, and all sum types are together taken away. Note the rule is obtained by reading the typing rule $(\mathsf{In}^{\downarrow})$ backward. The linear output rule is precisely dual.

$$(\mathsf{Out}^{\uparrow}) \ \frac{-}{\vdash_{\mathtt{O}} \overline{x}_i(\vec{y})P \triangleright \Gamma, \oplus_i x_i : (\vec{\tau}_i)^{\uparrow} \quad \stackrel{\overline{x_i}(\vec{y})}{\longrightarrow} \quad \vdash_{\mathtt{I}} P \triangleright \Gamma, \vec{y} : \vec{\tau}_i}$$

After the output at x_i is done, we no longer need the typing for x_i and other related linear channels, which are together taken off. Again the transition rule is backward-reading of the typing rule $(\mathsf{Out}^{\uparrow})$, restricted to the case of prime output. For a replicated input, the rule is straightforward. Below we set, for simplicity, each τ_i in $\vec{\tau}$ has **?**-mode while each ρ_j in $\vec{\rho}$ has $\uparrow$-mode.

$$(\mathsf{In}^{!}) \ \frac{-}{\vdash_{\mathtt{I}} !x(\vec{y}\vec{z}).P \triangleright \Gamma, x : (\vec{\tau}\vec{\rho})^{!} \quad \stackrel{x_i(\vec{y}\vec{z})}{\longrightarrow} \quad \vdash_{\mathtt{O}} !x(\vec{y}\vec{z}).P \mid P \triangleright \Gamma, x : (\vec{\tau})^{!}, \vec{y} : \vec{\tau}, \oplus_j y_j : \tau_j}$$

Note the channel x and its typing remain in the resulting process. Dually for

replicated output (with the dual convention for $\vec{\tau}$ and $\vec{\rho}$):

$$(\mathsf{Out}^{?})\ \frac{-}{\vdash_{\mathtt{O}} \overline{x}(\vec{y}\vec{z})P \triangleright \mathbf{?} \uparrow \Gamma, x:(\vec{\tau}\vec{\rho})^{?}{}_{\mathtt{O}} \ \xrightarrow{x(\vec{y}\vec{z})}\ \vdash_{\mathtt{I}} P \triangleright \mathbf{?} \uparrow \Gamma, x:(\vec{\tau}\vec{\rho})^{?}, \vec{y}:\vec{\tau}, \&_j y_j:\rho_j}$$

We also have the composition rules, (Com), (Par) and (Res).

$$(\mathsf{Com})\ \frac{\vdash_{\mathtt{I}} P \triangleright \Gamma \ \xrightarrow{x(\vec{y})} \vdash_{\mathtt{O}} P' \triangleright \Gamma' \quad \vdash_{\mathtt{O}} Q \triangleright \Delta \ \xrightarrow{\overline{x}(\vec{y})} \vdash_{\mathtt{I}} Q' \triangleright \Delta'}{\vdash_{\mathtt{O}} P|Q \triangleright \Gamma \odot \Delta \ \xrightarrow{\boldsymbol{\tau}}\ \vdash_{\mathtt{O}} (\boldsymbol{\nu}\, \vec{y})(P'|Q') \triangleright (\Gamma' \odot \Delta')/\vec{y}}$$

In (Par) below, we say Θ *allows* l when neither (1) $\Theta(x) = \updownarrow$ and either $l = x(\vec{y})$ or $l = \overline{x}(\vec{y})$; nor (2) $\Theta(x) = \mathbf{!}$ and $l = \overline{x}(\vec{y})$.

$$(\mathsf{Par})\ \frac{\vdash_{\phi} P \triangleright \Gamma \ \xrightarrow{l}\ \vdash_{\phi'} P' \triangleright \Gamma' \quad \Gamma \asymp \Delta \quad \Gamma \odot \Delta \text{ allows } l}{\vdash_{\phi} P|Q \triangleright \Gamma \odot \Delta \ \xrightarrow{\boldsymbol{\tau}}\ \vdash_{\phi'} P'|Q \triangleright \Gamma' \odot \Delta}$$

In (Par) we require that the IO-mode ϕ is preserved after the parallel composition, indicating Q is in the input mode. In fact, if Q is in the output mode, either $P|Q$ or $P'|Q$ (or both) become the composition of two output moded processes. The condition "$\Gamma \odot \Delta$ allows l" prohibits the action to take place which is impossible in the typed environment: for example, if $\Gamma \odot \Delta$ contains $\mathbf{!}$-type at x, an $\mathbf{?}$-output cannot take place since it already exists inside and two occurrences of the same $\mathbf{!}$ channel cannot be combined in typed composition.

$$(\mathsf{Res})\ \frac{\vdash_{\phi} P \triangleright \Gamma \xrightarrow{l} \vdash_{\phi'} P' \triangleright \Gamma' \quad x \notin \mathsf{fn}(l)}{\vdash_{\phi} (\boldsymbol{\nu}\, x)P \triangleright \Gamma/x \xrightarrow{l} \vdash_{\phi'} P' \triangleright \Gamma'}$$

For brevity we often write $P \xrightarrow{l} P'$ omitting type information. We observe the following properties. The proofs follow [5] and are omitted.

Proposition 5.1 (i) (subject transition) *If* $\vdash_{\phi} P \triangleright \Gamma$ *and* $\vdash_{\phi} P \triangleright \Gamma \xrightarrow{l} \vdash_{\phi'} P' \triangleright \Gamma'$ *then* $\vdash_{\phi'} P' \triangleright \Gamma'$.

(ii) (transition and IO-modes) *Let* $\vdash_{\phi} P \triangleright \Gamma \xrightarrow{l} \vdash_{\phi'} P' \triangleright \Gamma'$. *Then (a) if* l *is an input,* $\phi = \mathtt{I}$ *and* $\phi' = \mathtt{O}$, *(b) if* l *is an output,* $\phi = \mathtt{O}$ *and* $\phi' = \mathtt{I}$, *and (c) if* $l = \boldsymbol{\tau}$, $\phi = \phi' = \mathtt{O}$.

(iii) (transition and reduction) *Let* $\vdash_{\phi} P \triangleright \Gamma$. *Then* $\vdash_{\phi} P \triangleright \Gamma \xrightarrow{\boldsymbol{\tau}} \vdash_{\phi} P' \triangleright \Gamma$ *iff* $P \longrightarrow P'$.

(iv) (determinacy, 1) *If* $\vdash_{\phi} P \triangleright \Gamma \xrightarrow{l} \vdash_{\phi'} P'_{1,2} \triangleright \Gamma_{1,2}$ *then* $P'_1 \equiv P'_2$ *and* $\Gamma_1 = \Gamma_2$.

(v) (determinacy, 2) *If* $\vdash_{\mathtt{O}} P \triangleright \Gamma \xrightarrow{l_{1,2}}$ *then either (a)* $l_1 = l_2 = \boldsymbol{\tau}$ *or (b)* $l_1 = \overline{x}(\vec{y})$ *and* $l_2 = \overline{x}(\vec{y}')$.

5.2 Typed Transition and Innocence

A surprising observation [5] is that typed transition sequences of sequential processes precisely obey the conditions used to define strategies in Hyland-Ong games, where justification is made explicit as binding. In the following we outline the essential points of this coincidence, reaching the characterisation by innocence. Since the proofs precisely follow those in [5] we omit them.

Write $\vdash_\phi P \triangleright \Gamma \stackrel{s}{\Longrightarrow} \vdash_\psi Q \triangleright \Delta$ for the standard weak transition, i.e. given a sequence $s = l_1..l_n$ of non-$\boldsymbol{\tau}$ actions, we write $\vdash_\phi P \triangleright \Gamma \stackrel{s}{\Longrightarrow} \vdash_\psi Q \triangleright \Delta$ when $\vdash_\phi P \triangleright \Gamma \Longrightarrow \stackrel{l_1}{\longrightarrow} \ \ldots \ \stackrel{l_n}{\longrightarrow} \vdash_\psi Q \triangleright \Delta$ where $\Longrightarrow \stackrel{\text{def}}{=} \stackrel{\tau}{\longrightarrow}^*$. Without loss of generality, we assume the standard bound name condition on these sequences, i.e. binding names are always distinct and disjoint from free names. By Proposition 5.1 (4,5) we observe:

Proposition 5.2 (IO-alternation) *If $\vdash_\phi P \triangleright \Gamma \stackrel{s}{\Longrightarrow}$ then s is IO-alternating, i.e. whenever $s = s_1 \cdot l_1 \cdot l_2 \cdot s_2$ then l_1 is input and l_2 is output or vice versa.*

We next introduce views analogous to those of Hyland-Ong games. First, a notation: given a sequence $l_1..l_n$ of non-$\boldsymbol{\tau}$ actions under the bound name convention, we write $l_i \frown_{\mathsf{b}} l_j$ when $\mathsf{fn}(l_j) \subset \mathsf{bn}(l_i)$, i.e. a binder in l_i binds the free subject of l_j.

Now let s be a sequence of non-$\boldsymbol{\tau}$ actions. Then the *output view of s*, denoted $\ulcorner s \urcorner^{\mathtt{O}}$, is given by the following induction. Below ε is the empty sequence.

$$
\begin{array}{rcll}
\ulcorner \varepsilon \urcorner^{\mathtt{O}} & = & \varepsilon & \\
\ulcorner s \cdot l_n \urcorner^{\mathtt{O}} & = & \ulcorner s \urcorner^{\mathtt{O}} \cdot l_n & l_n \text{ is output} \\
\ulcorner s_1 \cdot l_i \cdot s_2 \cdot l_n \urcorner^{\mathtt{O}} & = & \ulcorner s_1 \urcorner^{\mathtt{O}} \cdot l_i \cdot l_n & l_n \text{ is input and } l_i \frown_{\mathsf{b}} l_n \\
\ulcorner s \cdot l_n \urcorner^{\mathtt{O}} & = & l_n & l_n \text{ is input and } \mathsf{fn}(l_n) \cap \mathsf{bn}(s) = \emptyset
\end{array}
$$

Dually we define the input view $\ulcorner s \urcorner^{\mathtt{I}}$. Note these definitions precisely follow those of the player/opponent views in Section 4. The visibility is defined accordingly: the empty sequence ε is output visible, a non-empty sequence $s = l_1...l_n$ is output visible if (a) each proper prefix of s is output visible, (b) if l_n is an output and $l_i \frown_{\mathsf{b}} l_n$ then l_i is in $\ulcorner l_1...l_n \urcorner^{\mathtt{O}}$. Dually we define input visibility. A sequence is *visible* if it is both input and output visible.

Proposition 5.3 *Let $\vdash_\phi P \triangleright \Gamma \stackrel{s}{\Longrightarrow}$. If s is input visible then it is output visible.*

Let s be visible. Then s is *well-bracketing* if, whenever $s' = s_0 \cdot l_i \cdot s_1 \cdot l_j$ for a prefix s' of s is such that (1) l_i is a question and (2) l_j is an answer free in $s_1 \cdot l_j$, we have $l_i \frown_{\mathsf{b}} l_j$.

Proposition 5.4 *If $\vdash_\phi P \triangleright \Gamma \stackrel{sl}{\Longrightarrow}$, l is output and s is well-bracketing, then*

sl is well-bracketing.

A weak transition sequence is *legal* if it is visible and well-bracketing. By determinacy, the bisimilarity (and other branching equivalences) and the trace equivalence on legal sequences coincide. We write the induced equivalence $\approx_{\mathsf{seq}}$.

Proposition 5.5 $\approx_{\mathsf{seq}}$ *is a congruence.*

Finally we observe:

Proposition 5.6 (innocence) *Let* $\vdash_\phi P \triangleright \Gamma \stackrel{s_{1,2}}{\Longrightarrow}$ *such that each of* $s_{1,2}$ *ends with an input action. Then if* $\ulcorner s_1 \urcorner \equiv_\alpha \ulcorner s_2 \urcorner$ *and* $\vdash_\phi P \triangleright \Gamma \stackrel{s_1 l_1}{\Longrightarrow}$ *then* $\vdash_\phi P \triangleright \Gamma \stackrel{s_2 l_2}{\Longrightarrow}$ *such that* $\ulcorner s_1 \urcorner l_1 \equiv_\alpha \ulcorner s_2 \urcorner l_2$.

Thus the behaviour of a sequential process is precisely characterised by a partial function which maps, up to α-equality, each output views ending with an input to the next output action (if any), which we again call the *innocent function*. We write $\mathbf{inn}(\vdash_\phi P \triangleright \Gamma)$ or simply $\mathbf{inn}(P)$ with the typing implicit, for the innocent function of $\vdash_\phi P \triangleright \Gamma$. Immediately $\mathbf{inn}(P) = \mathbf{inn}(Q)$ iff $P \approx_{\mathsf{seq}} Q$.

5.3 From Processes to Categories

From the universe of typed processes, we read off those processes which live in Hyland-Ong games. We make the essential use of injective renaming on both action types and processes [20]. Write $\Gamma =_{\mathsf{p}} \Delta$ if Δ is the result of permuting names in Γ. Note $=_{\mathsf{p}}$ is the equivalence relation. We write $[\Gamma]_{=_{\mathsf{p}}}$ for the quotient set containing Γ. Note all action types in $[\Gamma]_{=_{\mathsf{p}}}$ have precisely the same structure: the difference is only on names at interaction points.

Definition 5.7 An action type Γ is *Hyland-Ong* if, for each $x \in \mathsf{dom}(\Gamma)$, (1) the mode of $\Gamma(x)$ is **!**; and (2) if ρ of mode $\downarrow$ occurs in $\Gamma(x)$ then $\rho = ()^{\downarrow}$, dually if ρ of mode $\uparrow$ occurs in $\Gamma(x)$ then $\rho = ()^{\downarrow}$. If Γ is Hyland-Ong, then $[\Gamma]_{=_{\mathsf{p}}}$ is called a π-*arena*. We let $\alpha, \beta, \ldots$ range over π-arenas.

Note a π-arena, when each channel type is expanded as a (possibly infinite) syntax tree, gives a forest such that, for each component tree of the forest:

(i) Each node of an odd height (resp. even height) is labelled by **!** or $\downarrow$ (resp. **?** or $\uparrow$), that is it is an input (resp. an output). In particular, a root node is labelled only by **!**.

(ii) If a node is labelled $\downarrow$ or $\uparrow$ then it has no child node.

Note these conditions precisely correspond to Definition 4.1, reading input (resp. output) as "opponent" (resp. "player") and **!**, **?** (resp. $\downarrow, \uparrow$) as questions (resp. answers). Note also π-arenas are clearly a proper subset of all possible action types.

We next consider processes. Since we consider processes modulo $\approx_{\mathsf{seq}}$, we write $p, q, \ldots$ for the equivalence classes of the (typed) processes modulo $\approx_{\mathsf{seq}}$. Note the (typed) operation such as $p|q$ makes sense because of Proposition 5.5. We often write $\vdash_\phi p \triangleright \Gamma$ to make the typing of p explicit. Finally $[p]_{=_\mathsf{p}}$ denotes, as before, the equivalence class of p modulo renaming.

Definition 5.8 Given π-arenas α and β, we say p *inhabits from* α *to* β iff $\vdash_{\mathtt{I}} p \triangleright \overline{\Gamma}, \Delta$ for some $\Gamma \in \alpha$ and $\Delta \in \beta$ such that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. If p inhabits from Γ to Δ then $[p]_{=_\mathsf{p}}$ is called a *π-strategy from α to β*. We let $F, G, \ldots$ range over π-strategies and write $F : \alpha \to \beta$ if F is from α to β.

Proposition 5.9 *The following data defines a category.*

- *Objects: π-arenas.*
- *Arrows from α to β: π-strategies from α to β.*
- *Composition of arrows: Given $F : \alpha \to \beta$ and $G : \beta \to \gamma$, we define $F\,;G$ as the π-strategy $[(\boldsymbol{\nu}\,\mathsf{fn}(\Delta))(p|q)]_{=_\mathsf{p}}$ where, for mutually disjoint $\Gamma \in \alpha$, $\Delta \in \beta$ and $\Theta \in \gamma$, we have $\vdash_{\mathtt{I}} p \triangleright \overline{\Gamma}, \Delta$, $p \in F$, $\vdash_{\mathtt{I}} q \triangleright \overline{\Delta}, \Theta$ and $q \in G$.*

We call the resulting category π^{seq}_{HO}.

Proof. Firstly, $F;G$ does give a π-strategy from α to γ by Proposition 5.5. Secondly, the associativity of ; is immediate from that of $|$, that is, given disjoint sets of names X and Y, we have:

$$(\boldsymbol{\nu}\, Y)((\boldsymbol{\nu}\, X)(p|q)|r) \;\approx\; (\boldsymbol{\nu}\, X \cup Y)(p|q|r) \;\approx\; (\boldsymbol{\nu}\, X)(p|(\boldsymbol{\nu}\, Y)(q|r)).$$

Finally we construct the identity. Define a (possibly infinitary) process $\mathsf{CC}^{(\vec{\tau}\vec{\rho})^!}\langle ab\rangle$ and $\mathsf{CC}^{(\vec{\tau})^\downarrow}\langle ab\rangle$ by the following construction:

$$\begin{array}{rcl} \mathsf{CC}^{(\vec{\tau}\vec{\rho})^!}\langle ab\rangle & = & !a(\vec{z}\vec{w}).\overline{b}(\vec{y}\vec{u})(\Pi_i \mathsf{CC}^{\overline{\tau_i}}\langle y_i z_i\rangle \mid \&_j \mathsf{CC}^{\overline{\rho_i}}\langle w_i u_i\rangle) \\ \mathsf{CC}^{(\vec{\tau})^\downarrow}\langle ab\rangle & = & a(\vec{z}).\overline{b}(\vec{y})\Pi_i \mathsf{CC}^{\overline{\tau_i}}\langle y_i z_i\rangle \end{array}$$

For infinitary types, these recursions are meant to define the shape of a process at each height, so that the process is defined uniquely even for infinitary cases. For Γ which is Hyland-Ong, we may write $\mathsf{CC}\langle\Gamma, \Gamma'\rangle$ for the parallel composition of copy-cat agents for each prime action type, where Γ' is the renaming variant of Γ such that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$.

Now fix a π-arena α and let $\Gamma, \Gamma' \in \alpha$ such that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. We then set:

$$\mathbf{id}_\alpha \stackrel{\text{def}}{=} [\mathsf{CC}^\tau\langle\Gamma, \Gamma'\rangle]_{=_\mathsf{p}, \approx_{\mathsf{seq}}}.$$

We can then check that, for each P from Γ to Δ such that $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Delta) = \emptyset$ without loss of generality,

$$(\boldsymbol{\nu}\, \vec{x})(P|\Pi_i \mathsf{CC}\langle x_i y_i\rangle) \approx_{\mathsf{seq}} P\{\vec{y}/\vec{x}\}$$

where $\mathsf{dom}(\Gamma) = \{\vec{x}\}$ and $\mathsf{dom}(\Gamma') = \{\vec{y}\}$, hence as required. □

We can now conclude:

Theorem 5.10 *π^{seq}_{HO} and $\mathbb{CA}$ are categorically equivalent.*

Proof. By construction we already know π^{seq}_{HO} is a subcategory of $\mathbb{CA}$. Moreover the inclusion is subjective in objects up to isomorphism (under the well-ordering axiom) by appropriately indexing the nodes in an arena, at each height and at each tree. Thus it suffices to check the fullness. For this purpose we construct a concrete typed process which realises an arbitrary innocent behaviour. Firstly, an *abstract transition* is defined solely by type information as follows:

$$\vdash_{\mathtt{I}} \bullet \triangleright \Gamma, \Delta \stackrel{x_i(\vec{y}_i)}{\longrightarrow} \vdash_{\mathtt{O}} \bullet \triangleright \Gamma, \vec{y}_i : \vec{\tau}_i \qquad (\Delta = \&_i x_i : (\vec{\tau}_i)^{\downarrow})$$

$$\vdash_{\mathtt{O}} \bullet \triangleright \Gamma, \Delta \stackrel{\overline{x}_i(\vec{y}_i)}{\longrightarrow} \vdash_{\mathtt{O}} \bullet \triangleright \Gamma, \vec{y}_i : \vec{\tau}_i \qquad (\Delta = \oplus_i x_i : (\vec{\tau}_i)^{\uparrow})$$

$$\vdash_{\mathtt{I}} \bullet \triangleright \Gamma, x : (\vec{\tau}\vec{\rho})^{!} \stackrel{x(\vec{y}\vec{z})}{\longrightarrow} \vdash_{\mathtt{O}} \bullet \triangleright \Gamma, x : (\vec{\tau})^{!}, \vec{y} : \vec{\tau}, \oplus_i \vec{z} : \vec{\rho}$$

$$\vdash_{\mathtt{O}} \bullet \triangleright \Gamma, x : (\vec{\tau})^{?} \stackrel{\overline{x}(\vec{y}\vec{z})}{\longrightarrow} \vdash_{\mathtt{O}} \bullet \triangleright \Gamma, x : (\vec{\tau})^{?}, \vec{y} : \vec{\tau}.\&_i z_i : \rho_i$$

An *abstract innocent process* Ψ *under* Γ, ϕ is a prefix-closed set of abstract transition sequences which are (visible, well-bracketing and) innocent. Since for any arrow f in $\mathbb{CA}$ between π-arenas, we can find an abstract innocent process which realises f sequence by sequence up to renaming, it suffices to construct a (concrete) process whose weak transition coincides with Ψ. For this purpose we define P as the tree characterised by a relation between its node(s) and the corresponding transition in the output views in Ψ (for example, if it has an linear input as in the initial rule above, we can decide one component of a process as $\&_i x_i(\vec{y}_i).P_i$ with each P_i characterised by the subsequent transitions in the same way; if, in the output mode, no output is defined we use a diverging process of an appropriate type). Since each tree is at most of countable height, this uniquely characterises a syntax tree. □

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full Abstraction for PCF. *Info. & Comp.* 163 (2000), 409-470.

[2] S. Abramsky, K. Honda, and G. McCusker. A Fully Abstract Game Semantics for General References. *LICS*, 334-344, IEEE, 1998.

[3] S. Abramsky and G. McCusker. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions, ENTCS, Vol.3, North Holland, 1996.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[5] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-Calculus, *TLCA01*, LNCS 2044, pp.29-45, Springer, 2001.

[6] M. Berger, K. Honda, and N. Yoshida. Genericity and the π-Calculus, *FoSSacs03*, LNCS, Springer, 2003.

[7] G. Berry and P. L. Curien. Sequential algorithms on concrete data structures *TCS*, 20(3), 265-321, North-Holland, 1982.

[8] P. L. Curien. Sequentiality and full abstraction. Proc. of *Application of Categories in Computer Science*, LNM 177, 86–94, Cambridge Press, 1995.

[9] P. L. Curien. Abstract Böhm Trees, *Mathematical Structure of Computer Science*, 8(6), 1998.

[10] P. L. Curien and H. Herbelin. Computing with Abstract Böhm Trees, *Proc. of Third International Symposium on Functional and Logic Programming*, World Scientific, 1998.

[11] M. Fiore and K. Honda. Recursive Types in Games: axiomatics and process representation, *LICS'98*, 345-356, IEEE, 1998.

[12] F. Gadducci and U. Montanari. The Tile Model. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press,

[13] P. Di Gianantonio, G. Franco and F. Honsell. Game semantics for untyped λ-calculus, LNCS, 1998.

[14] J.-Y.Girard. Linear Logic, *TCS*, Vol. 50, 1–102, 1987.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] C.A.R. Hoare. *Unified Theories of Programming*, 58pp. Computing Laboratory, Oxford University, 1994.

[17] K. Honda. Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.

[18] K. Honda. Composing Processes, *POPL'96*, 344-357, ACM, 1996.

[19] K. Honda. A Theory of Types for the π-calculus. Typescript, 113 pp, October, 1999. Available from: `http://www.dcs.qmw.ac.uk/~kohei/`.

[20] K. Honda. Elementary Structures for Process Theory (1): Sets with Renaming. MSCS, 2001.

[21] K. Honda and M. Tokoro. An object calculus for asynchronous communication. ECOOP'91, LNCS 512, 133–147, 1991.

[22] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *Proc. ICALP'97, Proceedings of 24th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 1256, pp.225–236, Springer-Verlag, July, 1997.

[23] K. Honda and N. Yoshida. A uniform type structure for secure information flow, *POPL'02*, ACM, 2002.

[24] M. Hyland and L. Ong. "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.

[25] M. Hyland and L. Ong. Pi-calculus, dialogue games and PCF, *FPCA'95*, ACM, 1995.

[26] O. Jensen and R. Milner. Bigraphs and Transistions. *POPL'03*, ACM, 2003.

[27] G. Kahn and G. Plotkin. Concrete Domains. Theoretical Computer Science, 1993 (originally appeared as report 338 of Inria-Laboria, 1978).

[28] A. D. Ker, H. Nickau and L. Ong. Innocent game models of untyped lambda calculus. *Theoretical Computer Science* 272, pp. 247-292, 2002.

[29] N. Kobayashi, B. Pierce, and D. Turner. Linear Types and π-calculus, *POPL'96*, 358–371, ACM Press, 1996.

[30] J. Laird. Full abstraction for functional languages with control, *LICS'97*, IEEE, 1997.

[31] O. Laurent. Polarized games, LICS 2002, 265-274, IEEE, 2002.

[32] G. McCusker. Games and Full Abstraction for a Functional Metalanguage with Recursive Types. Ph.D.thesis, Imperial College, 1998.

[33] J. Meseguer. Research Directions in Rewriting Logic. *Computational Logic.* NATO Advanced Study Institute Series F, Vol. 165, pp. 345-398, Springer-Verlag, 1999.

[34] R. Milner. Fully abstract models of typed lambda calculi. *TCS*, 4:1–22, 1977.

[35] R. Milner. A Calculus of Communicating Systems, LNCS 92, Springer, Berlin, 1980.

[36] R. Milner. Functions as Processes. *MSCS*, 2(2), 119–146, CUP, 1992.

[37] R. Milner. Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.

[38] R. Milner. Address at Bologna on receiving Honorary Degree from the University of Bologna, 1997.

[39] R. Milner. Bigraphical reactive systems: basic theory. Computer Laboratory Technical Report No. 523, University Cambridge, 2002.

[40] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, *Info. & Comp.* 100(1), pp.1–77, 1992.

[41] M. Nickau. Hereditarily Sequential Functionals, LNCS 813, pp.253–264, Springer-Verlag, 1994.

[42] L. Ong. Correspondence between operational and denotational semantics. *Handbook of Logic in Computer Science*, Vol 4, pp. 269–356, Oxford University Press.

[43] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes, *MSCS* 6(5):409–453, 1996.

[44] G. D.Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[45] D. Scott and C. Strachey. Towards a Mathematical Semantics for Computer Languages. *Computers and Automata*. Polytechnic Institute of Brooklyn Press, pp. 19-46, 1971.

[46] N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus, *LICS'01*, IEEE, 2001. The full version as MCS technical report, 2001-09, University of Leicester, 2001. Available at www.mcs.le.ac.uk/~nyoshida.

[47] N. Yoshida, K. Honda, and M. Berger. Linearity and Bisimulation, Proc. of 5th International Conference, Foundations of Software Science and Computer Structures (FoSSaCs 2002), LNCS 2303, pp.417–433, Springer, 2002.

A Appendix

A.1 Structural Congruence

The structural congruence $\equiv$ is the least congruence which includes the standard α-equality and the above equations.

- $P|\mathbf{0} \equiv P$, $P|Q \equiv Q|P$, $(P|Q)|R \equiv P|(Q|R)$.
- $(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P$ and, if $x \notin \mathsf{fn}(Q)$, $(\boldsymbol{\nu}\, x)(P|Q) \equiv (\boldsymbol{\nu}\, x)P|Q$.
- $\overline{x}(\vec{y})\overline{u}(\vec{v})P \equiv \overline{u}(\vec{v})\overline{x}(\vec{y})P$, $(\boldsymbol{\nu}\, z)\overline{x}(\vec{y})P \equiv \overline{x}(\vec{y})(\boldsymbol{\nu}\, z)P$ $(z \notin \vec{y})$ and, if $\{\vec{y}\} \cap \mathsf{fn}(Q) = \emptyset$, $\overline{x}(\vec{y})(P|Q) \equiv \overline{x}(\vec{y})P|Q$.

A.2 Proof of Proposition 3.2

For (1), the closure under $\equiv$ is by rule induction on the generation rules of $\equiv$, precisely following the same proof in [5]. For the closure under $\longrightarrow$, it suffices to show the statement for the two generation rules. The case for replication is the same as [5]. The interesting case is linear reduction, which involves choice. Assume we have

$$\vdash_{\mathsf{0}} \&_i x_i(\vec{y_i}).P_i \mid \overline{x}_i(\vec{y_i})Q \triangleright \Theta. \tag{A.1}$$

Our goal is to show $(\boldsymbol{\nu}\, \vec{y_i})(P_i|Q)$ has the same typing as this term. By rolling back (Weak) (which is always possible except for the initial rule (Zero)), it

loses no generality if we assume this sequent is inferred by (Par). Thus we assume,

$$\vdash_{\phi_1} \&_i x_i(\vec{y}_i).P_i \triangleright \Theta_1, \qquad \vdash_{\phi_2} \overline{x}_i(\vec{y}_i)Q \triangleright \Theta_2 \tag{A.2}$$

such that $\phi_1 \odot \phi_2 = \phi$ and $\Theta_1 \odot \Theta_2 = \Theta$. We can again safely assume these terms are inferred by $(\mathsf{In}^{\downarrow})$ and $(\mathsf{Out}^{\uparrow})$, respectively. Thus, for input, we can set $\Theta_1 = \Gamma_1, \Delta$ such that $\mathsf{dom}(\Delta) = \{\vec{x}\}$ and $\phi_1 = \mathtt{I}$. By $(\mathsf{In}^{\downarrow})$ we infer:

$$\vdash_{\mathtt{O}} P_i \triangleright \Gamma_1, \vec{y}{:}\vec{\tau} \tag{A.3}$$

For output, by (1) and (2) of Definition 3.1 and by noting $x_i \in \mathsf{dom}(\Theta_1) \cap \mathsf{dom}(\Theta_2)$, we have $\Theta_2 = \Gamma_2, \overline{\Delta}$ where $\overline{\Delta}$ denotes the pairwise dual of Δ and

$$(\Gamma_1 \odot \Gamma_2) \otimes \vec{x}{:}\updownarrow = \ \Theta, \tag{A.4}$$

as well as $\phi_2 = \mathtt{O}$. By $(\mathsf{Out}^{\uparrow})$ we have:

$$\vdash_{\mathtt{I}} Q \triangleright \Gamma_2, \vec{y}{:}\overline{\vec{\tau}}. \tag{A.5}$$

By (A.3) and (A.5) we are done.

A.3 Proof of Proposition 5.1

(1) is by observing the correspondence between the transition rules and the typing rules. (2) is direct from the transition. For (3), the "if" direction is immediate, while the "only if" direction is by showing $P \xrightarrow{\overline{x}(\vec{y})} P'$ implies $P \equiv \overline{x}(\vec{y})Q|R$ as well as the dual case, from which we can generate the corresponding reduction (note we are taking terms modulo $\equiv$ in defining the transition). (4) and (5) are easy inspection of each rule.