Ludics is a model for the finitary linear Pi-calculus

Claudia Faggian[1] and Mauro Piccolo[2]

[1] PPS, CNRS-Paris 7 `faggian@pps.jussieu.fr`
[2] Universitá di Torino - PPS `piccolo@di.unito.it`

Abstract. We analyze in game-semantical terms the finitary fragment of the linear π-calculus. This calculus was introduced by Yoshida, Honda, and Berger [NYB01], and then refined by Honda and Laurent [LH06]. The features of this calculus - asynchrony and locality in particular - have a precise correspondence in Game Semantics. Building on work by Varacca and Yoshida [VY06], we interpret π-processes in linear strategies, that is the strategies introduced by Girard within the setting of Ludics [Gir01]. We prove that the model is fully complete and fully abstract w.r.t. the calculus.

1 Introduction

In this paper we show a precise correspondence between the strategies of Ludics [Gir01] and the linear π-calculus [NYB01]. Ludics has been introduced by Girard as an abstract game semantical model; the strategies, which here we call linear strategies, can be seen as a refinement of Hyland-Ong innocent strategies [HO00]. The linear π-calculus is a typed version of asynchronous π-calculus introduced by Yoshida, Honda and Berger. We interpret π-processes of the finitary fragment of the calculus into linear strategies. We prove that the model is fully complete w.r.t. the calculus.

Our analysis makes explicit an exact correspondence between process calculi features and game-semantical notions, in particular between asynchrony and innocence. Moreover, the names discipline of Ludics exactly matches that of the internal π-calculus.

The linear π-calculus: The linear π-calculus has been introduced by N. Yoshida and K. Honda in order to study strong normalization [NYB03], information flow security [KHY00,YH05], and other interesting properties of the calculus. The typing is based on Linear Logic.

The typing has recently be refined by Honda and Laurent [LH06], which establish a precise correspondence with polarized proof-nets. Hence typed $\pi-$calculus, Linear Logic, proof-nets, linear strategies fit together as aspects of the same broader picture.

The main features of the linear π-calculus are the following (for an in-depth discussion of these aspects, and their significance in concurrent and distributed computation, we refer to [SW01]):

Asynchrony: the act of sending a datum and the act of receiving it are separate. In particular no process need to wait to send a datum.

Internal mobility: only private (*fresh*) names can be communicated (by an output action).

Locality: names received in input are only used as output (and dually names sent in output are only used for input).

Linearity: The linearity constraint gives a discipline over the use of names. In particular it states that each linear name must be used at most once in the process.

In this paper, we study the **finitary** fragment of linear π-calculus, i.e. the fragment of linear π that does not contain the recursion and replication operators.

Ludics is a Game model, developed as an abstraction of Linear Logic. The strategies are a linear version of Hyland-Ong innocent strategies.

A key role is played by the notion of name (address), which play the same role as that of process calculus channel in an internal calculus (Sangiorgi's $\pi - I$). As we observed in previous work [FP06], the discipline on names imposed in [Gir01] is closely related to that of internal π-calculus.

Some other features which are specific to the Ludics setting - and fundamental for our interpretation of πprocesses - is the existence of 'incomplete' strategies, which terminate with an error state, and the treatment of the additives (corresponding to a prefixed summation).

Here we only use the basic level of the Ludics setting. Here we do not explore the full architecture, which however appears of great interest in the study of process calculi. In fact, Ludics: (i) comes equipped with a build-in notion of observational equivalence, (ii) has an interactive definition of types.

We expect that our work can open the way for applying the general setting of Ludics to a semantical analysis of process calculi. On the other direction, we hope to have an insight, making possible to import techniques concerning parallel execution and non-determinism which are well developed in the study of concurrency. In current work (see Section 5), we are exploring a possible extension of linear strategies with non-determinism.

Strategies and processes In this paper, we highlight the following correspondence:
asynchrony – innocence (which we discuss in Section 4.1)
internal mobility – names discipline in Ludics
locality – alternating arena.

2 The calculus

In this section we describe the finitary fragment of the linear π-calculus ([NYB01], [LH06]). We first discuss the untyped syntax and then introduce the typed setting.

2.1 Syntax and reduction rules

The syntax of the finitary linear π-calculus is given by the following grammar:

$$P ::= \; u(\boldsymbol{x}).P \mid \overline{u}(\boldsymbol{x})\, P \mid P|P \mid \nu\boldsymbol{x}\, P \mid 0$$

where is imposed that

internal mobility only private (fresh) names can be passed by an output action.

locality in $a(\boldsymbol{x}).P$ the names $\boldsymbol{x}$ are distinct and cannot be used as subject of an input action (and dually for $\overline{a}(\boldsymbol{x})\, P$).

Observe the two constructs (blocking input and asynchronous output) that mark the *asynchrony* of the calculus. The input $u(\boldsymbol{x}).P$ is blocking, i.e. the process waits for some input from the environment. The output $\overline{u}(\boldsymbol{x})\, P$ is the binding *asynchronous output* construct, defined in [NYB01]. The encoding in the standard π-calculus is the following:

$$\overline{u}(\boldsymbol{x})\, P =_{def} \nu\boldsymbol{x}(\overline{u}\langle\boldsymbol{x}\rangle|P)$$

This means that the continuation P can evolve in an independent way with respect to the output. If the output has no continuation (since the environment does not answer to it), we write $\overline{u}(\boldsymbol{x})$.

in $u(\boldsymbol{x}).P$ (and dually, $\overline{u}(\boldsymbol{x})\, P$) the names $\boldsymbol{x}$ are distinct bf ???and cannot be used as a subject of an input (resp. output) action.

The **input** $(u(\boldsymbol{x}).P)$ is blocking, i.e. means that the process waits for some input from the environment before continuing.

The **output** $(\overline{u}(\boldsymbol{x})\, P)$ is *asynchronous* and *binding* (as defined in [NYB01]). The encoding in the standard π-calculus is the following:

$$\overline{u}(\boldsymbol{x})\, P =_{def} \nu\boldsymbol{x}\,(\overline{u}\langle\boldsymbol{x}\rangle|P)$$

This means that the continuation P can evolve in an independent way with respect to the output.

Operational semantics. Reduction rules and structural congruence are those defined in [NYB03]. The main reduction rule is the following:

$$u(\boldsymbol{x}).P|\overline{u}(\boldsymbol{x})\, Q \longrightarrow \nu\boldsymbol{x}\,(P|Q) \tag{1}$$

which corresponds to the consumption of an asynchronous message by a receptor. To ensure the asynchrony of the output, the following rule is also added

$$P \longrightarrow P' \Rightarrow \overline{u}(\boldsymbol{x})\, P \longrightarrow \overline{u}(\boldsymbol{x})\, P' \tag{2}$$

Structural congruence is defined in term of the standard rules, extended with the following axioms (which allow to infer interaction under a prefixing output)

$$\overline{x}(\boldsymbol{u})\, \overline{y}(\boldsymbol{v})\, P \equiv \overline{y}(\boldsymbol{v})\, \overline{x}(\boldsymbol{u})\, P \;\text{ if } x \notin \boldsymbol{v} \text{ and } y \notin \boldsymbol{u} \tag{3}$$

$$\overline{x}(\boldsymbol{u})\,(P|Q) \equiv (\overline{x}(\boldsymbol{u})\, P)|Q \;\text{ if } \boldsymbol{u} \notin Q \tag{4}$$

$$\nu y\, \overline{x}(\boldsymbol{u})\, P \equiv \overline{x}(\boldsymbol{u})\, \nu y\, P \;\text{ if } y \notin \{x, \boldsymbol{u}\} \tag{5}$$

2.2 The typing system

We assume a countable set of names, ranged over by $u, v, x, y \dots$.

A **type** is assigned to a name in order to specify its use (the assignment is written $a : T$ where a is a name and T is a type). In particular, the capability a name can have, i.e. input, output or match, is specified by the polarity of the type: negative (T^-), positive (T^+), or neutral ($\updownarrow$). Moreover, the type of a name u disciplines also the data (names) which can be delivered using u. For example we write $u : \bigotimes_{i \in I} T_i$ to express that the channel u can be used in output mode to send an n-upla of names $\boldsymbol{x}$ where each x_i has type T_i. The syntax for the types is the following:

$$\begin{array}{rll}
T ::= T^+ \text{ positive} & |T^- \quad \text{negative} & |\updownarrow \text{ neutral} \\
T^+ ::= \mathbf{0} \quad \text{send error} & |\bigotimes_{i \in I} T_i^- \text{ output channel} & \\
T^- ::= \top \quad \text{non-reception} & |\bigparr_{i \in I} T_i^+ \text{ input channel} &
\end{array}$$

Given a type T, its **dual** **($T^\perp$)** is defined as $\top^\perp = 0$ and $(\bigparr_{i \in I} T_i)^\perp = \bigotimes_{i \in I} (T_i)^\perp$.

A **type environment** (denoted by the letters $\Gamma, \Delta, \dots$) gives a well-formed judgement on processes. It is a list of *distinct* names, each with a type assignment:

$$\Gamma = x_1 : T_1, \dots, x_n : T_n$$

A type environment Γ can be thought of as a partial function from names to types. Hence we write $Dom(\Gamma)$ for the set of names that occur in Γ; if $\Gamma = x : T, \Delta$, we have $\Gamma(x) = T$. With a slight abuse of notation, given a type environment Γ we denote by $\Gamma = \Gamma^+, \Gamma^\updownarrow, \Gamma^-$ its partition in positive, negative and neutral types.

The following operation on type environments is introduced in order to put the environments together when performing parallel composition of processes. Intuitively, this operation takes the union of two type environment and matches the names which appear in both environments.

Definition 21 ($\odot$-operator) *Let Γ and Δ be two type environments such that, for all $x \in Dom(\Gamma) \cap Dom(\Delta)$ we have $\Gamma(x) = \Delta(x)^\perp$. $\Gamma \odot \Delta$ is the environment Ξ such that $Dom(\Xi) = Dom(\Gamma) \cup Dom(\Delta)$ and*

$$\Xi(x) = \begin{cases} \Gamma(x) & \textit{if } x \notin Dom(\Delta) \\ \Delta(x) & \textit{if } x \notin Dom(\Gamma) \\ \updownarrow & \textit{otherwise} \end{cases}$$

Lemma 21 *$\odot$ is a partial commutative associative operator on type environments.*

Typing rules. Typing judgements are in the form $P \triangleright \Gamma$ and the corresponding typing rules are given in 1. The (ZERO)-rule types 0, that is the termination signal given by the process to the environment: it can be viewed as an error

$$\frac{}{0\triangleright}\ (\text{ZERO}) \qquad \frac{}{\triangleright a : \top}\ (\text{TOP}) \qquad \frac{P \triangleright \Gamma \quad \pi(T) \in \{+, \updownarrow\}}{P \triangleright \Gamma, x : T}\ (\text{WEAK})$$

$$\frac{P \triangleright x_1 : T_1^-, \ldots, x_n : T_n^-, \Gamma}{\bar{u}(\boldsymbol{x})\ P \triangleright u : \bigotimes_{i \in I} T_i^-, \Gamma}\ (\text{POS}) \qquad \frac{P \triangleright x_1 : T_1^+, \ldots, x_n : T_n^+, \Gamma^+}{u(\boldsymbol{x}).P \triangleright u : \bigparr_{i \in I} T_i^+, \Gamma^+}\ (\text{NEG})$$

$$\frac{P \triangleright \Gamma \quad Q \triangleright \Delta}{P|Q \triangleright \Gamma \odot \Delta}\ (\text{PAR}) \qquad \frac{P \triangleright \Gamma, x : T \quad \pi(T) \in \{-, \updownarrow\}}{\nu x\, P \triangleright \Gamma}\ (\text{RES})$$

Table 1. Typing rules for the finitary π-calculus

state or a lack of answer by the process to a question given by the environment. The (TOP)-rule, on the other hand, types the environment inaction, i.e a lack of answer by the environment to a question given by the process: note that in the process $\overline{u}(x)$ we have the continuation empty (the empty space), and it corresponds to the environment inaction on the channel x or a lack of answer on channel x). (NEG) and (POS) ensure the linearity constraint on names. Note the polarity constraint given in the rule (NEG): this ensures the decomposition of a given process into sub-processes with one negative port (see proposition **??**). The (PAR)-rule ensures the well-definiteness properties of the parallel composition. The (RES)-rule allows that only negative and neutral name can be restricted since they carry actions which expect their dual action to exist in the environment.

With respect to the given typing rules, we have the following results:

Proposition 21 (subject congruence) *If $P \triangleright \Gamma$ and $P \equiv Q$, then $Q \triangleright \Gamma$.*

Proposition 22 (subject reduction) *If $P \triangleright \Gamma$ and $P \longrightarrow Q$, then $Q \triangleright \Gamma$.*

Note. The purpose of the rule (TOP) is to state the name a to which the type $\top$ is associated.

In this sense, it is a way to report the environment inaction on a given channel x: it means that the environment knows the channel x but it does not use it. A more rigorous way to express this would be to annotate the negative inaction type with a name ($\top_x$), and making the rule (POS) more complicated.

The environment inaction should not be confused with process termination, which we denoted with 0.

Properties of the typing system. The typing system guarantee the following properties:

Proposition 23 *Let $P \triangleright \Gamma$. Then*

linearity: *for all $x \in Dom(\Gamma)$ such that $\Gamma(x) \neq \updownarrow$, x occurs at most once in P.*

subject reduction: $P \longrightarrow^* Q$ *then* $Q \triangleright \Gamma$
strong confluence: $P \longrightarrow Q_i$ *($i = 1, 2$) then either* $Q_1 \equiv Q_2$ *or there exists* R *such that* $Q_i \longrightarrow R$

2.3 The additive structure

We (sketch how to) extend the calculus with two new constructs: the branching input and the selective output[3] (in Linear Logic, these constructs correspond to the additive connectives; in a functional programming language, they correspond respectively to the case and to the select constructs).

The *syntax* is the following:

$$P ::= \ a \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i \mid \overline{a} in_j(\boldsymbol{x}_j)\, P_j \mid P|P \mid \nu \boldsymbol{x}\, P \mid 0$$

The construct $a \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i$ is the **branching input**. It corresponds to an external choice (a choice by the environment). We may see a as a multi-port channel: the process waits the environment to choose one of the components, and then evolves in the corresponding branch P_i. The reduction rule is

$$a \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i | \overline{a} in_j(\boldsymbol{x}_j)\, Q_j \to \nu \boldsymbol{x}_j P_j | Q_j$$

Two different branches P_i and P_j represent different evolutions of the system, which are in *conflict.*

The construct $\overline{a} in_j\, P_j$ is the **selective output,** which corresponds to an internal choice. If we want to have determinism, we need to impose that this choice is unique.

The positive and negative *types* are now:
$T^+ ::= \mathbf{0}$ zero $\quad \mid \bigoplus_{i \in I} \bigotimes_{k \in K_i} T_k^-$ selective output
$T^- ::= \top$ inaction $\mid \bigwith_{i \in I} \bigparr_{k \in K_i} T_k^+$ branching input

The *typing rules* are as in table (1), where we replace the (NEG) and (POS) rules with the following ones:

$$\frac{P \triangleright x_{1j} : T_{1j}, \ldots, x_{nj} : T_{nj}, \Gamma \quad j \in I}{\overline{u} in_j(\boldsymbol{x}_j)\ P_j \triangleright u : \bigoplus_{i \in I} \bigotimes_{k \in K_i} T_{ki}, \Gamma}\ pos \qquad \frac{\forall i. P_i \triangleright x_{1i} : T_{1i}, \ldots, x_{ni} : T_{ni}, \Gamma^+}{u \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i \triangleright u : \bigwith_{i \in I} \bigparr_{k \in K_i} T_{ki}, \Gamma^+}\ neg$$

All previous results (subject reduction, confluence, etc.) can be extended.

2.4 Refinements of the typing system.

The typing system can be refined by means of several constraints. In particular, Honda and Laurent establishes a hierarchy of classes of processes by adding constraints. The most relevant ones are acyclicity and sequentiality. We refer to [NYB01,NYB03,LH06] for the technical details.

[3] Our approach is closely inspired by [NYB03,VY06].

Acyclicity: a process is deadlock-free if it never reduces to stuck configurations (an example of stuck configuration is given by the process $\nu a, b(a.\overline{b}|b.\overline{a})$: here we have a *cyclic* dependency between a and b that blocks further reductions). The acyclicity constraint is added to control the way the resources are used by processes in order to avoid deadlocks. In [NYB03] acyclic processes are proved to be strongly normalizing.

Sequentiality: a process is sequential if at each step of reduction has at most one active output i.e. if there are multiple open inputs, the process can answer to only one of them. In [NYB01] sequential processes are used to provide a fully abstract encoding of PCF and in [LH06] sequential process are used to characterize polarized proof nets.

Important note. From now on *we assume the acyclicity constraint* in our typing rules since it guarantees the good behaviour of parallel composition. However, we *do not assume sequentiality.*

For this reason, in the model we use the variant of linear strategies which has been defined in [CF05] (there called **L-forests**) and which correspond to proves in Multiplicative Additive Linear Logic plus the Mix rule. If we add sequentiality, we exactly obtain the strategies defined in [Gir01] (there called **designs**).

DIRE L-FORESTS AND DESIGNS

3 The model

In this section we reformulate the definitions given in [Gir01,**?**]. We use the same notion of strategy as in Ludics, but types are here directly interpreted as arenas (as in [**?**]), to make more direct and explicit the construction defined in [Gir01]. The full interactive construction of types which belongs to Ludics would be possible -and very interesting- but it would be much less compact.

Our presentation is non-standard also in that: we use process calculus language to help the intuition. Moreover, we highlight the fact that a strategy is an event structure, and exploit the notion of conflict to describe the additive structure.

3.1 Name and actions

Let us consider an action in the sense of π-calculus: $u(\boldsymbol{x})$. To specify an action we need three data: the name u used as a channel, the set $\boldsymbol{x}$ of names that can be communicated on that channel, and the way the channel u can be used (input, output, etc.), i.e. its polarity. The same three data are specified by actions as defined in [Gir01]. Moreover, there is a coding, which is in many respect similar to that of De Bruijn notation.

Names and actions in Ludics. A notion which is crucial both in the π-calculus and in Ludics is that of *name* (in [Gir01] names are called *addresses*). A name can be seen as a channel, which is used to send or receive *data which are names themselves.*

Let the set $\mathcal{N}$ of the **names** (ranged over by $u, v, x, y, \dots$) be the strings of natural numbers. Given a name $u \in N$, the set $\{u.i | i \in \mathbb{N}\}$ corresponds to all data (names) that can be communicated using u.

This leads to an intuitive notion of action: an action is a pair (u, I), with $I \subset \mathbb{N}$, which specifies a name u used as channel and the set of names that will be communicated on that channel. By construction, to characterize such a set, it is enough to give the set of suffixes I that will be added to the name u.

Summing up, the action $a = (u, I)$ can communicate the names $u.i$, with $i \in I$. We write $name(a)$ for u.

Example. Consider the process $u(x, y).\bar{x}(z)$ We can use the following renaming: $x := u.1, y := u.2, z := u.1.1$ We write $u(x, y)$ as $(u, \{1, 2\})$, and $\bar{x}(z)$ as $(u.1, \{1\})$.

Polarities. A *polarized action* is an action a together with a polarity, positive (a^+) or negative (a^-), which specifies the capability of the name, that is if the channel is used for sending in output (positive) or receiving in input (negative).

Zero. The set of actions is extended with a special action, denoted by †, which indicates termination with an error. By definition, the action † is positive.

We will interpret the zero of process calculus with †, as 0 establishes the termination of the process. Intuitively, it states that player has no answer to an opponent move; in this senses it represents an *error* state.

Enabling relation. We say that the action $a = (u, I)$ generates the names $u.i$, written $a \vdash u.i$. Given two actions a, b, we write $a \vdash b$ if $a \vdash name(b)$. We call the relation $a \vdash b$ between actions *enabling* relation.

The enabling relation establishes dependency between the actions. This leads us to the notion of arena.

3.2 Arenas

In this section we define the notion of arena; arenas will interpret types. An arena is given by an *interface* and a *forest of polarized actions*, where the forest is induced by the enabling relation, which establishes dependency between the actions.

The interface of an arena specifies the names on which a strategy on that arena can communicate with the rest of the world. An **interface** $\mathcal{I}$ is a set of initial names $\mathcal{I} = \{u, v, \dots\}$ together with a polarity function $\pi : \mathcal{I} \to \{+, -\}$. We impose that the set of negative names is either empty or a singleton.

Given an interface $\mathcal{I}$, an **arena** on $\mathcal{I}$ is a set of polarized actions together with the enabling relation $b \vdash c$ (defined above), such that it satisfies the following conditions:

- the enabling relation is arborescent;

– The action c is minimal (denoted $\vdash c$) iff $name(c) \in \mathcal{I}$;
– the polarity of each minimal action c is that specified by the interface for $name(c)$; if $a \vdash b$ then the actions a and b have opposite polarity.

Given an arena of interface $\mathcal{I}$, we partition the trees according to the name of the root. If A is the set of all trees whose root uses the name u, we will write $u : A$. We will denote an arena of interface $\mathcal{I} = \{u_1, \ldots, u_k\}$ by $\mathit{\Gamma} = u_1 : A_1, \ldots, u_k : A_k$.

The polarity induces a partition of the names in the interface, and hence of the arena, into positive names (the outputs) and negative names (the inputs). With a slight abuse of notation, we write $\mathit{\Gamma} = \mathit{\Gamma}^+, \mathit{\Gamma}^-$. An arena is said *positive* if $\mathit{\Gamma}^-$ is empty (all names are positive), negative otherwise.

Arena constructions. We consider the following constructions on arenas.

Empty. The empty forest is an arena (positive or negative, according to the interface). We indicate the positive empty arena with $\mathit{\Gamma} : 0$ and the negative one with $u : \top$.

Dual. If $u : A$ is an arena, its dual $u : A^\perp$ is obtained by changing the polarity of u. Hence the actions are the same but the induced polarity is inverted.

Product. Let $\{u.i : A_i | i \in I\}$ be a family of arenas of negative (resp. positive) polarity. We define the arena $u : \prod_{i \in I} A_i$ by rooting all A_i on top of the action (u, I). If the polarity of u is positive (resp. negative), then we write $\prod_{i \in I} A_i = \bigotimes_{i \in I} A_i$ (resp. $\prod_{i \in I} A_i = \invamp_{i \in I} A_i$).

Sum. Let $\{u : A_i | i \in I\}$ be a family of arenas on the same interface and such that all roots are pairwise disjoint (hence, if (u, I) and u, J are roots of two distinct arenas, then $I \neq J$). We define the arena $u : \sum_{i \in I} A_i$ as the union of all the forests. If the polarity of the root is positive (resp. negative), then we write $\sum_{i \in I} A_i = \&_{i \in I} A_i$ (resp. $\sum_{i \in I} A_i = \bigoplus_{i \in I} A_i$).

3.3 Strategies

A strategy is here a forest of occurrences of actions. On the occurrences of actions is defined an order, which we denote by $\leq$. We write $e <_1 e'$ if the node e is immediate predecessor of e'.

Definition 31 (strategy) *Let $\mathit{\Gamma}$ be an arena. A strategy σ on the arena $\mathit{\Gamma}$ (written $\sigma : \mathit{\Gamma}$) is given by a forest $\langle E, \leq \rangle$, where E is a set of nodes, and $\leq$ is an arborescent partial order; the nodes are labelled by polarized actions[4] in $\mathit{\Gamma} \cup \{\dagger\}$; the polarity and the name of a node (written $\pi(e)$ and $name(e)$) are those of the labelling action. Formally, there is a labelling function $\lambda : E \to \mathit{\Gamma} \cup \{\dagger\}$ which satisfies the following conditions.*

[4] Hence nodes are occurences of actions.

justification: *For each node e, its labelling action is either initial ($\vdash \lambda(e)$) or there exits a preceding node $e' < e$ such that $\lambda(e') \vdash \lambda(e)$.*
innocence: *If $e <_1 e'$ and e is positive, then $\lambda(e) \vdash \lambda(e')$.*
positivity: *If e is maximal (i.e. there exists no e' such that $e < e'$), then e is positive. Moreover, if Γ is positive then the strategy is non empty.*

We denote the empty strategy with $\emptyset$ and the strategy whose unique action is a † with $\mathcal{D}ai$.

3.4 Linear strategies

In this section we describe linear strategies (as defined in [CF05]) which can be seen as an abstraction of Multiplicative-Additive Linear Logic with Mix.

Let us first introduce apart the definition of multiplicative strategy. This is a special case of linear strategy (which is enough to understand most of this paper). A strategy is **multiplicative** if no two labels use the same name. The more general definition below takes into account the repetitions due to the additive structure.

Let $\sigma : \Gamma$ be a strategy. We call **cell** a set of nodes that have the same name and the same predecessor. We call **positive** (resp. negative) a cell whose nodes are occurrences of positive (resp. negative) actions.

Starting from the notion of cell, we define on the nodes of a strategy the relation # of **conflict** as the smallest binary symmetric relation which satisfy the following conditions:

immediate conflict $e_1 \# e_2$ if they are distinct and belongs to the same cell;
inheritance if $k_1 < k_2$ and $k_1 \# k_3$ then $k_2 \# k_3$.

Definition 32 (linear strategy) *A strategy $\sigma : \Gamma$ is linear if it satisfies the following conditions*

linearity *for all distinct occurrences of actions k_1, k_2, if $name(k_1) = name(k_2)$ then $k_1 \# k_2$*
determinism *All non singleton cells are negative (i.e., every time there is a choice -expressed by the conflict relation- the choice belongs to Opponent).*

Observe that a linear strategy is multiplicative if all its cells are singletons, i.e. the conflict relation # is empty.

3.5 Totality

Typed processes will be interpreted into linear strategies which are total.

Definition 33 (totality) *A linear strategy $\sigma : \Gamma$ is total if, for each* negative *action c in the arena:*

1. *if $\vdash c$, then it occurs in σ (i.e. there is a node $e \in \sigma s.t.$ $\lambda(e) = c$);*
2. *if $b \vdash c$, each occurrence of b in σ is followed by an occurrence of c (i.e., for each $e \in \sigma$, $\lambda(e) = b \implies \exists e' \in sigma$ s.t. $e <_1 e'$ and $\lambda(e') = c$).*

3.6 Composition of strategies

Given two strategies σ_1, σ_2, we can compose them if they have compatible interfaces i.e. there is a common name that appear in both interfaces with opposite polarity.

A **cut net** is a finite set $\mathcal{R} = \{\sigma_1, \dots, \sigma_n\}$ of strategies such that (i) each name occurs at most in two interfaces, once as a positive name and once as a negative name, (ii) the graph which has as vertexes the interfaces and an edge connecting any two interfaces with a common name is acyclic.

The interface of the cut net is the interface induced by the names of the interfaces which are not a cut. For example, given a cut net whose strategies have interface u^+, a^+ and a^-, b^+, c^+ and b^-, d^+, the cut net has interface u^+, c^+, d^+.

We do not give details here on the composition of strategies, whose definition is given in [Gir01]. The result of composing the strategies in a cut-net $\mathcal{R}$ is called normal form of $\mathcal{R}$, which we denote by $\mathcal{R}^*$. This is a linear strategy having as interface the interface of the cut net.

3.7 Set of independent strategies

$\mathcal{S} = \{\sigma_i : \Gamma_i | i \in I\}$ is a set of **independent strategies** if (i) $\mathcal{D}ai \in \mathcal{S}$ and any two distinct strategies have disjoint interfaces. For example, two strategies $\sigma_1 : \{u^+, v^+\}$ and $\sigma_2 : \{z^-, t^+\}$ are independent, while two strategies with interfaces $\{u^+, v^+\}$ and $\{v^-, t^+\}$ are not.

A set of independent strategies $\mathcal{S} = \{\sigma_i : \Gamma_i | i \in I\}$ has interface $\Delta = \cup \Gamma_i$; we write $\mathcal{S} : \Delta$. **???**

Composition. Given two set $\Sigma = \{\sigma_i : \Gamma_i | i \in I\}, \Psi = \{\psi_j : \Delta_j | j \in J\}$ of independent strategies, we define their composition $\Sigma; \Psi$ as follows. Given $\mathcal{C} = \Sigma \cup \Psi$, we obtain a new set of independent strategies by partitioning the set $\mathcal{C}$ into cut-nets, according to the interfaces: we consider the graph whose vertices are the non-empty interfaces, and draw an edge between interfaces which contain dual names. The operation is only defined if the graph is acyclic, and each name appears at most in two distinct interfaces, with opposite polarity. The partition of the graph into connected components induces a partition of the strategies into cut nets $\mathcal{R}_1, \dots, \mathcal{R}_n$; we have $\Sigma; \Psi = \{\mathcal{R}_1^*, \dots, \mathcal{R}_n^*\}$, which is a set of independent strategies.

4 The interpretation

In this section , we interpret types with arenas, type environments with sets of arenas, and processes with linear strategies. For clarity, here we only show how to deal with the multiplicative case. To deal with the additive case is a straightforward extension (but the syntax becomes harder to read).

Interpretation of types.

$$\begin{aligned}
[\![u : \top]\!] &= u : \top \\
[\![\Gamma : \mathbf{0}]\!] &= \Gamma : 0 \\
[\![u : \bigotimes_{i\in I} T_i]\!] &= u : \bigotimes_{i\in I} [\![u.i : T_i]\!] \\
[\![u : \bigparr_{i\in I} T_i]\!] &= u : \bigparr_{i\in I} [\![u.i : T_i]\!]
\end{aligned}$$

Interpretation of type environments. The interpretation of a type environment is the juxtaposition of the interpretation of each type whose polarity is non neutral.

Interpretation of processes. A typed process is interpreted into a set of independent strategies. The most interesting case is that of a typed process $P \triangleright \Gamma$, where Γ has at most a negative type; this process will be interpreted into a strategy σ on the arena $[\![\Gamma]\!]$.

We use the following constructions on sets of independent strategies (see [CF05]) to give a semantics respectively to prefixed input, asynchronous output and the scope operator.

boxing: given a set Σ of independent strategies on $u.1 : T_1^+, \dots, u.n : T_n^+, \Gamma^+$ and a negative action $k = (u, I)^-$, the strategy $k.\Sigma : u : \bigparr_i T_i^+, \Gamma^+$ is obtained by prefixing the union of all strategies in Σ with k.

rooting: given a set Σ of independent strategies and a positive action $k = (u, I)^+$, the set of strategies $k \circ \Sigma$ is that obtained by adding a node of label (u, I), and making it precede only the nodes whose name is generated by (u, I) (where $(u, I) \vdash u.i$). Observe that, since the nodes of name $u.i$ are negative, they are roots. Hence we are prefixing with a node of label (u, I) all strategies on $u.i : T_i^-$.

restriction given a set of independent strategies Σ we obtain $\Sigma \setminus u$ by erasing all trees whose root has name u.

Let $P \triangleright \Gamma$. The interpretation is then defined in the following way

1. $[\![0 \triangleright \_]\!] = \{Dai\}$
2. $[\![\_ \triangleright x : \top]\!] = \{\emptyset, Dai\}$
3. $[\![\overline{u}(\boldsymbol{x})\, P \triangleright u : \bigotimes_{i\in I} T_i^-, \Gamma]\!] = (u, I) \circ [\![P[x_i := u.i] \triangleright u.1 : T_1^- \dots u.n : T_n^-, \Gamma]\!]$ where $\boldsymbol{x} = \langle x_1, \dots, x_n\rangle$ and $I = \{1, \dots, n\}$
4. $[\![u(\boldsymbol{x}).P \triangleright u : \bigparr_{i\in I} T_i^+, \Gamma^+]\!] = \{(u, I).[\![P[x_i := u.i] \triangleright u.1 : T_1^+ \dots u.n : T_n^+, \Gamma^+]\!], Dai\}$
5. $[\![\nu x\, P \triangleright \Gamma]\!] = [\![P \triangleright \Gamma, x : T]\!] \setminus x$
6. $[\![P_1 | P_2 \triangleright \Gamma \odot \Delta]\!] = [\![P_1 \triangleright \Gamma]\!]; [\![P_2 \triangleright \Delta]\!]$ supposing that $P_1 \triangleright \Gamma$ and $P_2 \triangleright \Delta$

4.1 Innocence and Asynchrony

An innocent strategy specifies what is Player answer to any Opponent move without having any information on the way in which Opponent plays. This means that after a Player move, we only know which Opponent moves are enabled, but we do not know if and in which order they will be played. Technically, each Opponent move immediately follows the Player move which enables it.

In our strategies, the causal order between actions takes into account the constraints which are given by the typing (the arena). Thus, for example, if the name x is generated by the action $u(x)$, any action using the channel x causally depends on $u(x)$ also in the strategy. Moreover, the strategy can introduce additional order. However, if the strategy is innocent, the extra order is only on pairs (Opponent move, Player move).

Innocence can be seen as the game-semantical counterpart of asynchrony and corresponds to the fact that in the asynchronous π-calculus, the input (Opponent move/negative action) is prefixing and blocking, while the output (Player move/positive action) is not. For output we use rooting: the only constraints we have are those fixed by the arena. Boxing instead introduces additional order.

The correspondence between innocent strategies and processes in an asynchronous π calculus appears clearly when analyzing the normal forms (section 4.4).

4.2 Full Completeness

Proposition 41 (Validity) *If $P \triangleright \Gamma$ is a process, its interpretation $[\![P]\!]$ on $[\![\Gamma]\!]$ is a set of independent strategies, where each strategy is total (on the corresponding arena).*

Proposition 42 (Correctness) *The interpretation satisfies the following.*
If $P \equiv Q$ then $[\![P]\!] = [\![Q]\!]$
If $P \longrightarrow Q$ then $[\![P]\!] = [\![Q]\!]$

Proposition 43 (Completeness) *Let $\Gamma = \Gamma^-, \Gamma^+$ be the arena interpreting a typing environment Γ^-, Γ^+, where Γ^- is either empty or a singleton. If σ is a linear strategy on that arena, and σ is total, then σ is the interpretation of a process $P \triangleright \Gamma^-, \Gamma^+$.*

We sketch the proof, only dealing with the multiplicative case.

Proof. We associate to the strategy the typing derivation of a process. The proof is by induction on the size $|\sigma|$ of σ, where the size of a strategy is the number of its actions which are not †.

$|\sigma| = 0$ *and σ negative.* Being negative, the interface contains a negative name, u, corresponding to the negative arena $u : A$. As σ is the empty strategy, for it to be total the arena $u : A$ must be empty. This case corresponds to the $SKIP$ rule (possibly followed by weakening).

$|\sigma| = 0$ *and σ positive.* We have that $\sigma = \mathfrak{D}ai$; this correspond to the zero rule (possibly followed by weakening).

$|\sigma| > 0$ *and* σ *negative.* This case correspond to the NEG rule. We have that $\sigma = (u, I).\sigma'$ is a total strategy on the arena $u : A, \Gamma^+$, where $A = \parr_I A_i$ and σ' is a total *positive* strategy on the arena $u.i : A_i, \ldots \Gamma^+$. To check totality, we observe that the Positivity condition implies that σ' is non empty.

$|\sigma| > 0$ *and* σ *positive.* We have that σ is a forest of positive strategies. Let us analyze each single connected component (each tree), on the opportune arena (the arena containing the names used by the strategy). Putting them together will correspond to the PAR rule (possibly followed by weakening).

Assume that $\sigma = (u, I). \cup_{i \in I} \sigma_i$. This is a strategy on the arena $u : A, \Gamma^+$, where $A = \otimes_I A_i$. All the addresses used in each σ_i are distinct, hence we can partition Γ^+ into $\Gamma_1^+, ..., \Gamma_k^+$ according to the names of the actions which are initial in each σ_i. Each σ_i is a total negative strategy on the arena $u.i : A_i, \Gamma_i^+$.

4.3 Full abstraction

By using the same technique as in [NYB03], we have the following result of full abstraction, where the notion of operational equivalence is the *typed weak bisimilarity* ($\approx$) defined in [NYB03].

Proposition 44 (full abstraction) $P \triangleright \Gamma \approx Q \triangleright \Gamma \iff [\![P \triangleright \Gamma]\!] = [\![Q \triangleright \Gamma]\!]$

Let us sketch the proof. All typed processes are strongly normalizing, hence we can define two processes to be equivalent if they reduce to the same normal form, up to structural congruence. Such an equivalence is the same as $\approx$, and we can take the (unique) term in normal form as canonical representative in each class of π-terms. The key result is the following lemma.

Lemma 41 *Let $P \triangleright \Gamma$ and $Q \triangleright \Gamma$ be in normal form. $P \approx Q$ if and only if $P \equiv Q$.*

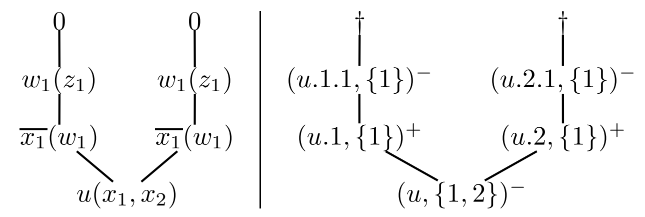
4.4 Normal forms

In [NYB03] is given a canonical characterization of normal forms: each term in normal form can be written as a parallel composition of sub-processes which have a Bohm-tree like structure. These trees exactly correspond to Ludics strategies.

Example. Let us consider two normalized terms which are structurally equivalent (both are typed on $\Gamma = u : (\otimes(\parr T_1^+)) \parr (\otimes(\parr T_2^+)))$:

$$u(x_1, x_2).\overline{x_1}(w_1)\,\overline{x_2}(w_2)\,w_1(z_1).0|w_2(z_2).0 \equiv u(x_1, x_2).\overline{x_1}(w_1)\,w_1(z_1).0|\overline{x_2}(w_2)\,w_2(z_2).0$$

By reordering the actions performed by the two processes according to the Labelled Transition System, we obtain the tree on the left hand side below. On the right hand side, instead, we have the interpretation as linear strategy on the right: there is an exact correspondence.



5 Discussion and future work

We have shown a precise correspondence between the finitary fragment of the linear π-calculus [NYB01,NYB03] and the linear strategies introduced by Girard in the setting of ludics [Gir01].

Building on this core, we aim at extending the calculus and the model with non-determinism, recursion and replication, by following the approach proposed by [VY06].

Moreover, it is possible to use the full architecture of Ludics, and in particular the interactive constructions on types, and we are interested in exploring also this direction.

Non determinism. In current work [**?**] we extend the calculus and the model with internal choice, i.e. with non determinism.

On one side, we add to the calculus a τ-prefixed sum ($\sum_i \tau.P_i$)

On the other side, the set of actions is extended with neutral actions, and a non deterministic choice is modeled by a cell of neutral actions. Non-determinism has the same behavious as the additive structure, but the actions labelling the cell are "silent".

Recursion and replication. In this paper we have worked a finitary version of the linear π-calculus, in order to establish a correspondence with the existing setting of Ludics, as defined in [Gir01]. However, we expect to be able to extend the model both wiht replication (by using techniques similar to those which are developped in [VY06]) and with recursion.

References

[CF05] P.-L. Curien and C. Faggian. L-nets, strategies and proof-nets. In *CSL 05 (Computer Science Logic)*, LNCS. Springer, 2005.

[FP06] C. Faggian and M. Piccolo. A graph abstract machine describing event structure composition. In *GT - VC, Graph Transformation for Verification and Concurrency*, 2006. ENTCS.

[Gir01] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11:301–506, 2001.

[HO00] M. Hyland and L. Ong. On full abstraction for PCF. *Information and Computation*, 2000.

[KHY00] V. Vasconcelos K. Honda and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, LNCS. Springer, 2000. Extended abstract.

[LH06] O. Laurent and K. Honda. An exact correspondence between a typed pi-calculus and polarised proof-nets. draft, 2006.

[NYB01] K. Honda N. Yoshida and M. Berger. Sequentiality and the pi-calculus. In *Proc. of TLCA 2001, the 5th International Conference on Typed Lambda Calculi and Applications*, LNCS. Springer, 2001. Extended abstract.

[NYB03] K. Honda N. Yoshida and M. Berger. Strong normalisation in the pi-calculus. *Journal of Information and Computation*, 2003. full version.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[VY06] D. Varacca and N. Yoshida. Typed event structures and the pi-calculus. In *MFPS*, 2006.

[YH05] N. Yoshida and K. Honda. Noninterference through flow analysis. *Journal of Functional Programming*, 2005. revised.