

Concurrent Game Semantics: Easy as Pi

Simon Castellan, Léo Stefanesco, Nobuko Yoshida

▶ **To cite this version:**

Simon Castellan, Léo Stefanesco, Nobuko Yoshida. Concurrent Game Semantics: Easy as Pi. [Research Report] Inria. 2020. hal-03128187

HAL Id: hal-03128187
https://inria.hal.science/hal-03128187

Submitted on 2 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

arXiv:2011.05248v1 [cs.PL] 10 Nov 2020

Game Semantics: Easy as Pi

Introducing Programming Game Semantics

SIMON CASTELLAN, Inria, Univ Rennes, IRISA, France
LÉO STEFANESCO, Collège de France, France
NOBUKO YOSHIDA, Imperial College London, United Kingdom

Game semantics has proven to be a robust method to give compositional semantics for a variety of higher-order programming languages. However, due to the complexity of most game models, game semantics has remained unapproachable for non-experts.

In this paper, we aim at making game semantics more accessible by viewing it as a syntactic translation into a session typed π-calculus, referred to as *metalanguage*, followed by a semantics interpretation of the metalanguage into a particular game model. The syntactic translation can be defined for a wide range of programming languages without knowledge of the particular game model used. Simple reasoning on the model (soundness, and adequacy) can be done at the level of the metalanguage, escaping tedious technical proofs usually found in game semantics. We call this methodology *programming game semantics*.

We design a metalanguage (π_{DiLL}) inspired from Differential Linear Logic (DiLL), which is concise but expressive enough to support features required by concurrent game semantics. We then demonstrate our methodology by yielding the first causal, non-angelic and interactive game model of $\mathsf{ML}_{\parallel}$, a higher-order call-by-value language with shared memory concurrency. We translate $\mathsf{ML}_{\parallel}$ into π_{DiLL} and show that the translation is adequate. We give a causal and non-angelic game semantics model using event structures, which supports a simple semantics interpretation of π_{DiLL}. Combining both of these results, we obtain the first interactive model of a concurrent language of this expressivity which is adequate with respect to the standard weak bisimulation, and fully abstract for the contextual equivalence on second-order terms.

We have implemented a prototype which can explore the generated causal object from a subset of OCaml.

Additional Key Words and Phrases: game semantics, π-calculus, session types, Linear Logic, event structures

1 INTRODUCTION

Background. Reasoning about programs requires a mathematical model of their execution, called semantics. A popular technique is *operational semantics* which models the concrete execution on the metal by an abstract machine made of mathematical symbols and Greek letters. Its popularity is due to its simplicity and flexibility, modelling a wide range of programming features. However, operational semantics only gives meaning to closed programs of ground types. *Open higher-order* programs, ie. programs with external functional parameters are left aside. This creates difficulties, for instance, comparing such programs can only be done through an untractable quantification over arbitrary contexts. Denotational semantics tackles this problem by trying to give *compositional semantics*, usually as a function from its external parameters to the result. The most common form of denotational semantics, based on domain theory, copes well with higher-order functions, but more laboriously with effectful computations, and almost not at all with concurrency.

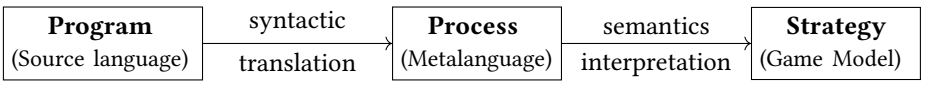
At the dawn of the 1990s, a new form of denotational semantics appeared: *game semantics* [Abramsky et al. 2000; Hyland and Ong 2000]. There, an open program is modelled by its possible **interactions** with the context. Thanks to its interactivity, this methodology has proved to be very extensible and easily supports a wide range of programming features (references, control operators, probabilities, concurrency, quantum, etc.)

Authors' addresses: Simon Castellan, Inria, Univ Rennes, IRISA, Rennes, France, simon@phis.me; Léo Stefanesco, Collège de France, Paris, France, leo.lveb@gmail.com; Nobuko Yoshida, Imperial College London, London, United Kingdom, n.yoshida@ic.ac.uk.

Moreover, as observed by Ghica and Tzevelekos [2012], game semantics reconciles denotational and operational semantics together: the interaction traces of a program can be computed either denotationally (by induction on the syntax) or operationally (by running an abstract machine).

While game semantics has been recognised as a powerful tool to build denotational models, we believe that its most useful and promising feature is the simplicity with which one can *describe* the compositional behaviour of systems in general (eg. with its recent use for verifying compilers [Koenig and Shao 2020; Stewart et al. 2015] or operating systems [Gu et al. 2018]). So far, its simplicity has not been apparent, and game semantics is often considered inaccessible to non-experts who wish to define a denotational model based on games for their favourite language.

The Framework. The thesis of this paper is that, the complexity of game semantics interpretations can be decomposed by introducing a simple message-passing intermediate language, between the source program and the model. Indeed, game semantics bundles two ideas together: the idea of interpreting a program as a process interacting with its environment, *and* a semantic interpretation of these processes. Our main contribution in this paper is to make this separation explicit by factorising the interpretation of a language in game semantics as the following steps:



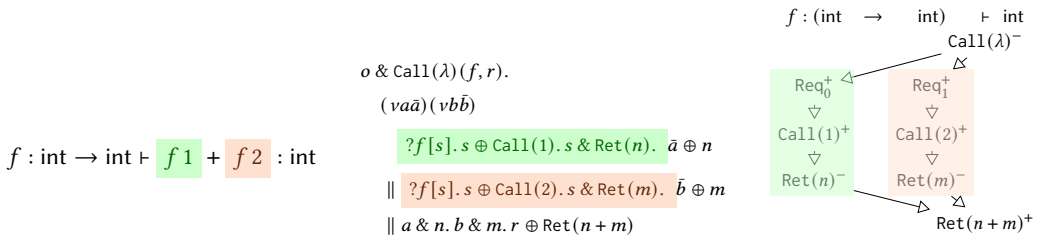
This factorisation offers several advantages. First, it allows to decouple the interpretation of the source language from the details of the model. One can use the same translation to obtain different models (eg. traces, event structures, LTSs, ...), and, conversely, one can interpret a language becomes a matter of writing a syntactic translation, an easier task than doing the interpretation from a language to a game model directly, and which becomes model-agnostic.

A Language for Strategies. What would a good intermediate language for strategies be? Strategies represent the interaction of the program with the environment in the form of messages that are exchanged between them. For that reason, a message-passing language such as the π-calculus [Milner et al. 1992] is an ideal candidate. Dating back to encodings of the call-by-name and call-by-value λ-calculi by Milner [1992a], the π-calculus has been used to encode a wide range of programming languages, including functional, concurrent, and distributed languages. Its connection with game semantics has been studied ever since the introduction of game semantics [Hyland and Ong 1995]. Some game semantics interpretations rely on the use of the categorical semantics of Linear Logic [Melliès 2009]. Recently, Caires and Pfenning [2010] have discovered a Curry-Howard correspondence between the π-calculus and Linear Logic through *session types*. Specifically, this paper uses an extension of the calculus in [Wadler 2014] to Differential Linear Logic (DiLL) [Ehrhard 2018] where $\otimes$ and $\parr$ are identified to be able to interpret languages with deadlocks. Session types are a natural fit here since their connection with game semantics have been recently discovered [Castellan and Yoshida 2019] (in a prototypical setting). Our metalanguage is going beyond other session type-based calculi to be able to express game semantics interpretations.

Contributions and Outline of the Paper. This paper proposes a uniform framework where we view game semantics interpretations as syntactic translations to a process calculus, followed by a semantic interpretation of this calculus into a game model.

We demonstrate this methodology using one translation and one semantics interpretation: we translate $\mathsf{ML}_{\parallel}$, which is a mini ML extended with shared memory concurrency into the metalanguage, π_{DiLL} based on Differential Linear Logic.[1] Our semantic interpretation of π_{DiLL} is based on

[1]It differs from $\pi\mathtt{DILL}$ (dual intuitionistic linear logic) in [Caires and Pfenning 2010].



(a) Program (in ML‖) (b) Process (in π_{DiLL}) (c) Strategy (event structure)

Fig. 1. An overview of the methodology on an example

an extension of concurrent games [Rideau and Winskel 2011], that deals with internal actions and replication. Combining the syntactic translation and semantics interpretation gives us the first causal and interactive interpretation of a concurrent call-by-value language, which is adequate – and fully-abstract for second-order interfaces – for a strong notion of contextual equivalence. This bridges the gap with real-world programming languages, as our implementation of the model allows to explore the causal behaviour of concurrent programs written in (a subset of) OCaml.

The methodology is illustrated in Figure 1 on a program calling an external function f twice in parallel, as function parameters are executed in parallel in this language. This program is translated to the π-calculus which is then interpreted as an event structure describing the causal relationships between the different actions of the program (see § 4 and § 5).

Our paper is structured as follows: **§ 2** presents an overview of the paper and introduces the source language ML‖; **§ 3** presents our metalanguage π_{DiLL} and its equational theory; **§ 4** presents the translation of ML‖ into π_{DiLL}; **§ 5** presents the causal model of the metalanguage; **§ 6** presents the interpretation of π_{DiLL} into the model; **§ 7** outlines our prototype implementation of a causal interpretation of a subset of OCaml. **§ 8** provides related work and **§ 9** concludes with future work. work. Proofs can be found in ***Appendix***, and we shall submit our prototype implementation to ***Artifact Evaluation***.

We make use of the `knowledge` package. **Definitions** of mathematical concepts appear blue boldface, and their uses occur in blue and are linked to the original definition.

2 OVERVIEW: GAME SEMANTICS AS MESSAGE-PASSING TRANSLATION

In this section, we informally introduce our methodology. In particular, we explain basic ideas behind game semantics interpretations, and their syntactic counterpart in our metalanguage. This section paves the way to the formal definition of π_{DiLL} in § 3. In § 2.1, we first introduce our source language, a concurrent call-by-value language, ML‖. We then illustrate different aspects of the language and the techniques we use to represent them: § 2.2 focuses on first-order terms; § 2.3 on higher-order behaviour; and finally § 2.4 discusses shared memory.

2.1 ML‖: Mini-ML with shared-memory concurrency

We use a concurrent call-by-value language with integer references, called ML‖:

$$\begin{array}{llll} \text{type} & \sigma, \tau & ::= & \bullet \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{ref} \mid \sigma \to \tau \qquad \text{value} \quad V ::= x \mid \lambda x.\, M \mid \underline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid () \\ \text{term} & M, N & ::= & V \mid M\, N \mid Y \mid \bot \mid \mathtt{if}\, M\, N\, N' \mid \mathtt{plus} \mid \mathtt{equal} \mid \mathtt{get} \mid \mathtt{set} \mid \mathtt{ref} \end{array}$$

A **base type** is either the unit type written $\bullet$, int or bool. Y is the fixpoint operator. We use standard syntactic sugar: $\mathtt{let}\, x = M \,\mathtt{in}\, N$ for $(\lambda x.\, N)\, M$; $\mathtt{let}\, f\, x = M \,\mathtt{in}\, N$ for $\mathtt{let}\, f = \lambda x.\, M \,\mathtt{in}\, N$. Note that our language does not include products or sum types for presentation purposes. It is straightforward to extend our approach to those, and our implementation supports them.

$$\frac{}{\Delta, x : \sigma \vdash x : \sigma}\,[\textsc{var}] \qquad \frac{\Delta, x : \sigma \vdash M : \tau}{\Delta \vdash \lambda x.\, M : \sigma \to \tau}\,[\textsc{abs}] \qquad \frac{\Delta \vdash M : \sigma \to \tau \quad \Delta \vdash N : \sigma}{\Delta \vdash M\, N : \tau}\,[\textsc{app}] \qquad \frac{}{\Delta \vdash \bot : \sigma}\,[\textsc{bot}]$$

$$\frac{}{\Delta \vdash Y : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau}\,[\textsc{fix}] \qquad \frac{}{\Delta \vdash \mathsf{tt}, \mathsf{ff} : \mathsf{bool}}\,[\textsc{bool}] \qquad \frac{\Delta \vdash M : \mathsf{bool} \quad \Delta \vdash N_1 : \sigma \quad \Delta \vdash N_2 : \sigma}{\Delta \vdash \mathtt{if}\, M\, N_1\, N_2 : \sigma}\,[\textsc{if}]$$

$$\frac{}{\Delta \vdash \underline{n} : \mathsf{int}}\,[\textsc{num}] \qquad \frac{}{\Delta \vdash \mathtt{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}}\,[\textsc{plus}] \qquad \frac{}{\Delta \vdash () : \bullet}\,[\textsc{unit}]$$

$$\frac{}{\Delta \vdash \mathtt{ref} : \mathsf{int} \to \mathsf{ref}}\,[\textsc{ref}] \qquad \frac{}{\Delta \vdash \mathtt{get} : \mathsf{ref} \to \mathsf{int}}\,[\textsc{get}] \qquad \frac{}{\Delta \vdash \mathtt{set} : \mathsf{ref} \to \mathsf{int} \to \bullet}\,[\textsc{set}]$$

Fig. 2. Typing system of $\mathsf{ML}_\parallel$

Instead of having specific constructs (eg. $M := N$), we chose to have higher-order constants which makes the metatheory simpler. Hence, we write $M+N$ for $\mathtt{plus}\, M\, N$; $M = N$ for $\mathtt{equal}\, M\, N$; $!M$ for $\mathtt{get}\, M$; $M := N$ for $\mathtt{set}\, M\, N$. We have not included any sequential or parallel composition operators in the language as it is definable in the language: $M; N$ is a shorthand for $(\lambda x.\, N)\, M$ when x not occurring free in N and $M \parallel N$ is a shorthand for $(\lambda xy.\, ())\, M\, N$: this behaves as expected due to our semantics of application, which evaluates arguments in parallel.

Typing of this language is standard, given by a judgement $\Delta \vdash M : \sigma$ where $\Delta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and is shown in Figure 2. The terms `equal`, `plus`, `get`, `set`, and `ref` are called **constants**. An **interface** is a pair of a context Δ and a type σ, written $\Delta \vdash \sigma$.

Operational Semantics. As it is customary for such languages mixing higher-order and shared state, we define the operational semantics in two steps: we first define a confluent reduction relation $\longrightarrow$ on all terms, which only reduces functional redexes, leaving reference operations unchanged. The rules for this reduction relation are standard and given in Figure 3a, where we use the following evaluation contexts: $E ::= [\,] \mid E\, M \mid M\, E \mid \mathtt{if}\, E\, M\, N$. This choice of contexts implies in particular that in $M\, N$ the evaluation of M and N are done *in parallel*: This allows for operators that evaluate their arguments in parallel; e.g. $M + N$, which unfolds to $\mathtt{plus}\, M\, N$, evaluates M and N in parallel. Sequentiality is only ensured between the body of a function and its argument: in $(\lambda x.\, M)\, N$, N is always evaluated first. Since $M \parallel N$ is simply defined as $(\lambda xy.())\, M\, N$, this also guarantees that M and N are evaluated in parallel.

In a second step, we look at terms of the form $\Delta \vdash M : \sigma$ where (1) σ is not a functional type; and $\Delta(a) = \mathsf{ref}$ for all $a \in \mathrm{dom}(\Delta)$. Such terms are called **semiclosed**, and are going to be executed by a machine whose states are tuples $\Delta \vdash^{\vec{y}} \langle M, \mu \rangle$ where $\Delta \vdash M : \sigma$ is semiclosed, $\vec{y}$ is a subset of $\mathrm{dom}(\Delta)$, the *public* locations; and $\mu : \mathrm{dom}(\Delta) \to \mathbb{N}$ is the memory state, mapping locations to values. The type σ is the type of the machine $\Delta \vdash^{\vec{y}} \langle M, \mu \rangle$.

We chose to give the operational semantics of such machine states under the form of a labelled transition system. The visible actions will correspond to memory actions described by $\Sigma_{\mathsf{ML}_\parallel} ::= \mathsf{r}\langle x, k \rangle \mid \mathsf{w}\langle x, k \rangle$ where x is a free reference variable and $k \in \mathbb{N}$ is a value: $\mathsf{r}\langle x, k \rangle$ means the program has read value k from x, and $\mathsf{w}\langle x, k \rangle$ that it has written k to x. The transitions of the LTS are labelled over the set $\Sigma_{\mathsf{ML}_\parallel} \cup \{\tau\}$, which is ranged over by α, β. The silent action τ is often omitted. The LTS of $\mathsf{ML}_\parallel$ is given in Figure 3b. Note that $\vec{y}$ stays unchanged through out the reduction.

Lemma 2.1 (Subject Reduction). (1) *If* $\Delta \vdash M : \sigma$ *and* $M \longrightarrow N$, *then* $\Delta \vdash N : \sigma$; *and* (2) *if* $\Delta \vdash M : \sigma$ *is semiclosed and* $\Delta \vdash^{\vec{y}} \langle M, \mu \rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}} \langle M', \mu' \rangle$ *then* $\Delta' \vdash M : \sigma$.

Weak bisimulation and observation equivalence. We define the standard weak bisimulation relation between two machines as equivalence on base terms. By the LTS rules in Figure 3, weak transitions are defined as expected: we write $\Longrightarrow$ for the reflexive, transitive closure of $\xrightarrow{\tau}$, $\overset{\alpha}{\Longrightarrow}$ for transitions $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and $\overset{\hat{\alpha}}{\Longrightarrow}$ for $\overset{\alpha}{\Longrightarrow}$ if $\alpha \neq \tau$ and $\Longrightarrow$ otherwise.

$$\frac{V \text{ a value}}{(\lambda x.\, M)\, V \to M[V/x]}\,[\beta] \qquad \frac{}{\underline{n} + \underline{m} \to \underline{n+m}}\,[\text{SUM}] \qquad \frac{}{\underline{n} = \underline{n} \to \text{tt}}\,[\text{ET}] \qquad \frac{n \neq m}{\underline{n} = \underline{m} \to \text{ff}}\,[\text{EF}]$$

$$\frac{}{\texttt{if tt}\, M\, N \to M}\,[\text{IFT}] \qquad \frac{}{\texttt{if ff}\, M\, N \to N}\,[\text{IFF}] \qquad \frac{V, V' \text{ are values}}{Y\, V\, V' \to V\,(Y\, V)\, V'}\,[\text{Y}] \qquad \frac{M \to N}{E[M] \to E[N]}\,[\text{CONTEXT}]$$

(a) Deterministic reduction for functional fragment of ML‖

$$\frac{M \to N}{\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\tau} \Delta \vdash^{\vec{y}} \langle N, \mu\rangle}\,[\text{BASE}] \qquad \frac{x \in \text{dom}(\Delta) \quad x \in \vec{y} \Rightarrow \alpha = \text{w}\langle x, n\rangle \quad x \notin \vec{y} \Rightarrow \alpha = \tau}{\Delta \vdash^{\vec{y}} \langle x := \underline{n}, \mu\rangle \xrightarrow{\alpha} \Delta \vdash^{\vec{y}} \langle (), \mu[x := n]\rangle}\,[\text{WRITE}]$$

$$\frac{\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle}{\Delta \vdash^{\vec{y}} \langle E[M], \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle E[N], \mu'\rangle}\,[\text{CXT}] \qquad \frac{x \in \text{dom}(\Delta) \quad x \in \vec{y} \Rightarrow \alpha = \text{r}\langle x, \mu(x)\rangle \quad x \notin \vec{y} \Rightarrow \alpha = \tau}{\Delta \vdash^{\vec{y}} \langle !x, \mu\rangle \xrightarrow{\alpha} \Delta \vdash^{\vec{y}} \langle \underline{\mu(x)}, \mu\rangle}\,[\text{READ}]$$

$$\frac{r \notin \text{dom}(\Delta)}{\Delta \vdash^{\vec{y}} \langle \texttt{ref}(\underline{n}), \mu\rangle \xrightarrow{\tau} \Delta, r : \text{ref} \vdash^{\vec{y}} \langle r, \mu[r := n]\rangle}\,[\text{ALLOC}]$$

(b) Labelled Transition System for ML‖ semiclosed terms

Fig. 3. Operational semantics of ML‖

Definition 2.2 (Weak bisimulation of ML‖ machines). A relation $\mathcal{R}$ over machines of type $\bullet$ is a *weak bisimulation* if

(1) for all $\Delta_1 \vdash^{\vec{y}} \langle M_1, \mu_1\rangle\ \mathcal{R}\ \Delta_2 \vdash^{\vec{y}} \langle M_2, \mu_2\rangle$, whenever $\Delta_1 \vdash^{\vec{y}} \langle M_1, \mu_1\rangle \xrightarrow{\alpha} \Delta_1' \vdash^{\vec{y}} \langle M_1', \mu_1'\rangle$, there exists $\Delta_2 \vdash^{\vec{y}} \langle M_2, \mu_2\rangle \xRightarrow{\hat{\alpha}} \Delta_2' \vdash^{\vec{y}} \langle M_2', \mu_2'\rangle$ such that $\Delta_1' \vdash^{\vec{y}} \langle M_1', \mu_1'\rangle\ \mathcal{R}\ \Delta_2' \vdash^{\vec{y}} \langle M_2', \mu_2'\rangle$.
(2) The symmetric case of 1.

The largest such bisimulation is called weak bisimilarity, denoted by $\approx_{\text{ML}\|}$.

We define the *observational equivalence* on terms by quantifying over machine contexts.

Definition 2.3 (Observational equivalence of ML‖). Given $\Delta \vdash M, N : \sigma$, M and N are observationally equivalent, written $M \cong_{\text{ML}\|} N$ when for all machine contexts C of $\bullet$ type (that is a machine of $\bullet$ type with a hole in the term component), $C[M] \approx_{\text{ML}\|} C[N]$.

2.2 First-order programs

We start our investigation of the interactive semantics of ML‖ with the first-order case. An interface $\Delta \vdash \sigma$ is **first-order** when σ is a base type and Δ only contains base types. In call-by-value, such terms have a limited interaction with their environment, as variables stand for values: waiting for the values of its parameters from the environment, computing the result, and finally possibly returning it to the environment (or diverging).

For each type σ, we define a set $\mathfrak{C}(\sigma)$ of messages describing values of type σ. For base types, $\mathfrak{C}(\sigma)$ is exactly the set of values of σ:

$$\mathfrak{C}(\text{int}) = \mathbb{N} \qquad \mathfrak{C}(\bullet) = \{()\} \qquad \mathfrak{C}(\text{bool}) = \{\text{tt}, \text{ff}\}.$$

For non-base types, as we will see in the next section, this correspondence will not hold. This extends to context by letting $\mathfrak{C}(\Delta)$ be the set of tuples $\prod_{a \in \text{dom}(\Delta)} \mathfrak{C}(\Delta(a))$. So, the protocol of a first-order interface $\Delta \vdash \sigma$ is very simple: *Program* expects a tuple of constructors for its parameters (an element of $\mathfrak{C}(\Delta)$) and then may return its result, a constructor of σ.

Unsurprisingly, our language of session types must then include type constructors representing communication from *Context* to *Program* and from *Program* to *Context*. Since there are only two participants, we choose to describe the protocol from the point of view of *Program*: the first kind

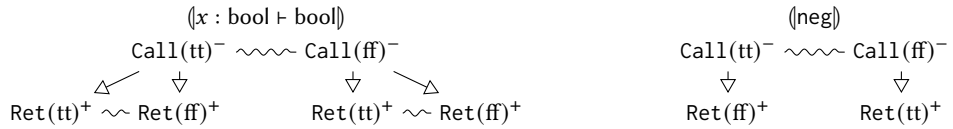


Fig. 4. Semantics interpretation of the negation: game (left) and strategy (right)

of messages will be seen as *inputs* (and be assigned a negative polarity) while the second kind will be seen as *outputs* (and assigned a positive polarity).

We use here a dialect of session types inspired by Linear Logic, and this data flow is captured by the *additive connectives*: & (input) and ⊕ (output). Concretely, the translation of a first-order interface $\Delta \vdash \sigma$ gives the following protocol: $\langle\!|\Delta \vdash \sigma|\!\rangle = \&_{i \in \mathfrak{C}_{(\Delta)}} \mathtt{Call}(i)^- \cdot \oplus_{o \in \mathfrak{C}_{(\sigma)}} \mathtt{Ret}(o)^+$: *Program* must input a message $\mathtt{Call}(i)$ containing the parameter values and may output a message $\mathtt{Ret}(o)$ with its return value.

The interpretation of a first-order type is simply the process that waits for the input parameter, computes the value and returns it. This process is parametrised by a name o representing the communication channel with *Context*. For instance, the negation is:

$$\langle\!|x : \mathsf{bool} \vdash \mathtt{neg} \stackrel{\text{def}}{=} \mathtt{if}\ x\ \mathtt{ff}\ \mathtt{tt} : \mathsf{bool}|\!\rangle_o = o \,\&\, \{\mathtt{Call(tt)}(r).\, r \oplus \mathtt{Ret(ff)};\ \ \mathtt{Call(ff)}(r).\, r \oplus \mathtt{Ret(tt)}\}.$$

It starts by an input on o and there are two branches since we can receive two possible values. Along with the value received, there is also a new name (r) which is used for the continuation of the protocol: in this case, returning the result. (We will see soon that in this particular case it is possible to reuse the name o, as is customary in session types, but in general it is not possible.)

In the semantic world, the session type $\langle\!|\Delta \vdash \sigma|\!\rangle$ becomes a **game** (where moves are messages), and the process $\langle\!|\Delta \vdash M : \sigma|\!\rangle$ becomes a **strategy**, which in our setting explicits the causal relationships between the different actions of the program (in ML$_\parallel$: memory operations, calls and returns). Both games and strategies are described by the same mathematical object: an **event structure**, which is a partial order (written $\rightarrowtriangle$) along with a conflict relation ($\sim$). In the example of the negation, we have drawn both the game and the strategy in Figure 4. Conflict represents incompatibility within an execution: two events in conflict will occur in different executions. In games, this represents the fact that the protocol forbids to play these two moves in the same execution, corresponding to choices (& and ⊕) in the syntax of session types: *Context* has to choose a set of parameters, and *Player* a return value. Note that in this simple case, the strategy is simply a subset of the game.

2.3 Higher-order programs

This might seem a lot of complexity to capture simple input/output behaviours. However, it becomes justified when looking at higher-order programs where the control jumps between *Program* and *Context* due to calling external functions, i.e.. functions of the interface. Let us start with a simple (second-order) example $f : \bullet \to \bullet \vdash M := f(); f() : \bullet$. As before, *Context* starts by sending a constructor for each parameter. But what should be a constructor for the type $\bullet \to \bullet$? In game semantics, we explicitly observe function calls and returns across the interface. To model this, *Context* sends a token, which we write simply λ, instead of sending a function or some code. However, now the session between *Program* and *Context* is split in two independent sub-sessions: one, as before, for *Program* to return the result, and a second one to make calls to f. To represent this splitting of a session into several sub-sessions, we use the notation $\|$, which represents the

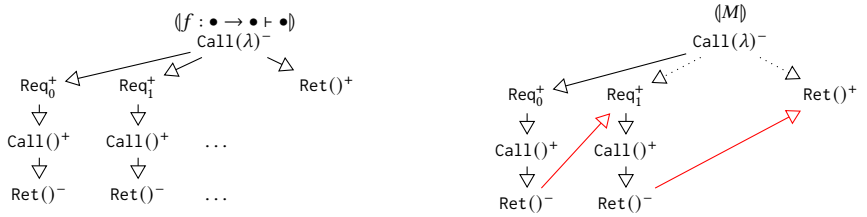


Fig. 5. Interpretation of higher-order terms

multiplicative connectives of Linear Logic ($\otimes, \parr$) identified into one.[2] The interface thus becomes:

$$\llbracket f : \bullet \to \bullet \vdash \bullet \rrbracket = \texttt{Call}(\lambda)^- \cdot (\texttt{Call}()^+ \cdot \texttt{Ret}()^- \parallel \texttt{Ret}()^+).$$

where we use $\texttt{Call}(\lambda)^- \cdot S$ instead of the unary choice $\&_{e \in \{*\}} \texttt{Call}(\lambda)^- \cdot S$ and similarly for $\texttt{Call}()^+ \cdot S$.

In this protocol, *Program* receives the λ token from *Context*, and then can either call f or return. If *Program* calls f, then it might return (). However, there is some information that is not captured here. *Program* should be allowed to call f several times, but may only return only once: certain sessions can be repeated while some others may not. To encode this information in the session type, we use the standard exponentials of Linear Logic: by default every message is linear (can only occur once). Then, we have two dual modalities that allow to specify duplication: $?S$ means *Program* can spawn as many instances of S as they like; while $!S$ means *Context* can spawn as many instances of S as they like, hence *Program* must behave as a server accepting requests from *Context*.

Our example requires $?(\cdot)$ since *Program* decides how many times f is called. We get:

$$\llbracket f : \bullet \to \bullet \vdash \bullet \rrbracket = \texttt{Call}(\lambda)^- \cdot (?(\texttt{Call}()^+ \cdot \texttt{Ret}()^-) \parallel \texttt{Ret}()^+).$$
$$\llbracket M \rrbracket_o = o \,\&\, \texttt{Call}(\lambda)(f, r).\, ?f[s].\, s \oplus \texttt{Call}()[r_1].\, r_1 \,\&\, \texttt{Ret}().$$
$$?f[s].\, s \oplus \texttt{Call}()[r_2].\, r_2 \,\&\, \texttt{Ret}().\, r \oplus \texttt{Ret}()$$

The above example features two new constructs. First, due to $\parallel$, it is possible to introduce several names (in the input on o), as there might be several sub-sessions now: for instance, f denotes here the session related to f while r that related to returning the result. Second, we have the construction $?f[s]$ which opens a new session (of type S) on f called s when f has type $?S$. In this case, sessions on name f are used to make a call to parameter f. Note that we can reuse the name s across sessions as, being a linear name, it disappears after the $s \oplus \texttt{Call}()$. The interpretation of M and its interface are given in Figure 5. The game is now more complex: we see clearly the possible independent sub-sessions as being incomparable for the causal order. The $?S$ construction is putting infinitely many copies of S, each prefixed by a positive move $\texttt{Req}_i$ equipped with a copy index $i \in \mathbb{N}$. When the strategy plays, it chooses indices for the positive moves: this choice is irrelevant (two different choices will yield equivalent strategies). In this case, the strategy is still a subset of the game, but there are now extra causal links indicating the sequentiality specific to the term (arrows in red).

By duality, the ! construction is used when *defining* functions. To construct a process implementing a session type $!S$, we use a construct called **promotion**, written $!a(x).\, P$, which represents a

[2]The distinction in Linear Logic is crucial to ensure deadlock-freedom and hence cut-elimination. Since programs of ML$\parallel$ do have deadlocks, we explicitly want to identify the multiplicative connective.

server spawning up P each time a request is made, with x being the request. For instance, we have:

$$(\!|\vdash \bullet \to \bullet|\!) = \mathtt{Call}()^- \cdot (\lambda^+ \cdot !(\mathtt{Call}()^- \cdot \mathtt{Ret}()^+))$$
$$(\!|\lambda x.()|\!)_o = o \,\&\, \mathtt{Call}().\, o \oplus \lambda.\, !o(c).\, c \,\&\, \mathtt{Call}().\, c \oplus \mathtt{Ret}().$$

A function starts by telling *Context* that it does normalise to a λ-abstraction, by sending λ, and then sets up a server to handle calls to the function by *Context*. In this term, we have not specified a list of names for some actions: this means we simply reuse the subject of the action, for instance $o \oplus \lambda$ stands for $o \oplus \lambda[o]$.

2.4 Shared memory concurrency

To conclude our illustration of the phenomena at play in the game semantics of a language like ML$_\parallel$, we now outline the case of references. Our viewpoint is that each reference has an owner – the one that declared it with `ref`, and when an open program has a free variable of type ref, it is not the owner of the reference, but *Context* is. Operations on non-owned references are simply forwarded to the owner: if a term M has a free reference r, then what it receives from *Context* is simply a name at which it can send the memory operations. In our simple setting, there are only two operations: `get` and `set`. For instance, $r : \mathsf{ref} \vdash r := !r+1 : \bullet$ will be interpreted as a process that does two requests on the object r: a `get`, and then a `set`. Dually, when interpreting the construct $\mathtt{ref}(\cdot)$, *Program* is the owner, and *Context* is the user (since the reference is returned to *Context*): this means that *Program* must create a server that can adequately handle these requests.

However, such a server cannot be implemented using the promotion described in § 2.3. Indeed, such servers are *stateless*, meaning that the answer to a request does not depend on previous or parallel requests. In particular, two independent requests can (and are) treated independently. This is not the case for a server implementing a reference, which has to treat requests in a sequential order. Two independent request must thus be sequentialised. To represent this phenomenon, we turn our attention to an extension of Linear Logic, called *Differential Linear Logic* [Ehrhard 2018], which extends Linear Logic with such servers. We use here a slight variant of the standard *codereliction*, which allows to set up a one-time server. Such "servers", written $\#a(x).\, P$ in our metalanguage introduced in the next section, wait for *one* request x, and execute P on it. P has still access to a so it can start a new such server if it wants to, after having processed the request x. In the meanwhile, the server is down, and requests are left pending. In particular two concurrent requests are racing to be satisfied by this server. This phenomenon induces a natural sequentialisation of the requests on a. The implementation of this reference server is given in § 4.2.

3 π_{DiLL}: A METALANGUAGE FOR STRATEGIES OF GAME SEMANTICS

In this section, we present a formal treatment of the intuitions presented in § 2, and formalise it into a calculus that we call π_{DiLL}, as its typing rules are close to *Differential Linear Logic* (DiLL). In § 3.1 we present the types of π_{DiLL}; in § 3.2 we present the syntax of the calculus for session types and processes along with the typing discipline. In § 3.3 we define the operational semantics of π_{DiLL}; and in § 4 we present the interpretation of ML$_\parallel$ into π_{DiLL}.

3.1 Types of π_{DiLL}

As explained in § 2, our types closely mirror the syntax of Linear Logic formulas:

$$T ::= \&_{i\in I} \ell_i^- \cdot T_i \mid \oplus_{i\in I} \ell_i^+ \cdot T_i \mid (T \parallel \ldots \parallel T) \mid ?S \mid !S.$$

These types describe protocols from the point of view of *Program*. The first two are the *additive* connectives and represent data input and data output respectively: $\&_{i\in I} \ell_i^- \cdot T_i$ means that *Program*

should expect a message consisting in one of the ℓ_i from *Context* and then behave as the corresponding T_i; and dually $\oplus_{i\in I}\ell_i^+ \cdot T_i$ indicates that *Program* should output to *Context* one of the ℓ_i and continue as the corresponding T_i. In both instances, I is an arbitrary countable set of choices; and ℓ_i represents the **payload** of the message, to be thought of simple datatypes such as integers or booleans choosen in a set $\mathbb{L}$. When I is a singleton we write $\ell^- \cdot S$ and $\ell^+ \cdot S$ directly.

The construct $T_1 \parallel \ldots \parallel T_n$ is the parallel composition and represents both $\parr$ and $\otimes$: in this setting with deadlocks, we do not need to distinguish between the two multiplicatives connectives of Linear Logic and can identify them (as in compact-closed models of Linear Logic). From a protocol standpoint, this means that the current session splits into n independent subsessions. In Session Types, these two connectives are called input and output respectively and represents delegation, or channel-passing. However, in this setting closer to the internal π-calculus [Sangiorgi 1996] where free names are never sent, the right intuition is that channels are not passed around, but messages occurring in one of the subsession is just prefixed with the index of the sub-session (thus encoding the so-called *multiplicative addresses*), and the names are just abstraction around these addresses. A parallel composition can be empty, leading to the type **1**, which is the equivalent of the (identified) multiplicative identities of Linear Logic.

The dual types $!S$ and $?S$ are the exponentials and represent replicable sessions: $!S$ means that *Program* is expected to receive arbitrarily many queries from *Context* to start a session of type S (ie. *Program* is the server, and *Context* the client); while $?S$ means that *Program* can contact *Context* to start as many sessions of type S as it wants (ie. *Program* is a client and *Context* the is a server).

Types come equipped with a notion of *duality* stemming from the usual De Morgan laws: & and $\oplus$ are dual to each other, as well as ? and ! while $\parallel$ and **1** are self-dual. Session types whose toplevel constructors are not $\parallel$ are called **rooted** as they have a unique minimal action. Only such types will be bound to channels in π_{DiLL}.

3.2 Processes of π_{DiLL}

The syntax of π_{DiLL} follows the notation of [Wadler 2014].

$$P, Q ::= \mathbf{0} \mid P \parallel Q \mid (\nu ab)P \mid c \,\&\, \{\ell_i(\vec{x}_i).\, P_i\}_{i\in I} \mid c \oplus k[\vec{x}].\, P \mid\ !c(\vec{x}).\, P \mid \#c(\vec{x}).\, P \mid ?c[\vec{x}].\, P$$

$\mathbf{0}$ denotes a nil process, $P \parallel Q$ is a *parallel* composition, $(\nu ab)P$ is a *restriction* where binding a and b in P. $c \,\&\, \{i(\vec{x}_i).\, P_i\}_{i\in i}$ is a **branching** with labels $\ell_i \in \mathbb{L}$ indexed by $i \in I$ where I can be infinite; its dual, $c \oplus k[\vec{x}].\, P$, is a *selection* to label k. **promotion** $!c(\vec{x}).\, P$ (stateless server), **codereliction** $\#c(\vec{x}).\, P$ (stateful server), and **dereliction** $?c[\vec{x}].\, P$ (their client) have been explained in § 2. In all constructs, $\vec{x}$, $\vec{x}_i$ are lists of bound names as in the internal π-calculus [Sangiorgi 1996].

We consider the finite or **infinite** processes generated by this grammar. Concretely, we consider the set of processes to be the ideal completion of the finite processes ordered by the subtree ordering. This is standard in some literature of the π-calculus (eg. [Castagna and Padovani 2009]) and necessary to give a syntactic counterpart to infinite strategies which are used in the interpretation of most languages, for recursion and memory. Definition of such infinite processes will mostly be by computing the least fixpoint of continuous functions. Such fixpoints are written $\mathrm{fix} f$.

Typing rules. We now describe the typing rules for the calculus. Typing contexts are of the form $\Gamma := a_1 : A_1, \ldots, a_n : A_n$ where each A_i is a rooted type. This is key since we want every action of our π-calculus to correspond to a meaningful computational event.

While typing judgements are flat lists, session types have more structure since $(A \parallel B) \parallel C$ is only *equivalent* to $A \parallel (B \parallel C)$ (ie. isomorphic as games as will be seen in § 6.1), but not equal. This prompts us to define a way to flatten a non-rooted session type into a context. For that, we define

$$\frac{}{\mathbf{0} \triangleright \Gamma}\ [\text{NIL}] \qquad \frac{P \triangleright ?\Gamma, \Gamma_1 \quad Q \triangleright ?\Gamma, \Gamma_2}{P \parallel Q \triangleright ?\Gamma, \Gamma_1, \Gamma_2}\ [\text{PAR}] \qquad \frac{P \triangleright \Gamma, a : A, b : A^\perp}{(\nu ab)P \triangleright \Gamma}\ [\text{RES}] \qquad \frac{P \triangleright ?\Gamma, \vec{x} :: T}{!a(\vec{x}).\, P \triangleright ?\Gamma, a : !T}\ [\text{REP}] \qquad \frac{P \triangleright \Gamma, a : ?T, \vec{x} :: T}{?a[\vec{x}].\, P \triangleright \Gamma, a : ?T}\ [\text{REQ}]$$

$$\frac{\forall i \in I, P_i \triangleright \Gamma, \vec{x}_i :: T_i}{a \,\&\, \{\ell_i(\vec{x}_i).\, P_i\}_{i \in I} \triangleright \Gamma, a : \&_{i \in I} \ell_i^- \cdot T_i}\ [\text{BR}] \qquad \frac{k \in I \quad P \triangleright \Gamma, \vec{x} :: T_k}{a \oplus \ell_k[\vec{x}].\, P \triangleright \Gamma, a : \oplus_{i \in I} \ell_i^+ \cdot T_i}\ [\text{SEL}] \qquad \frac{P \triangleright \Gamma, a : !T, \vec{x} :: T}{\#a(\vec{x}).\, P \triangleright \Gamma, a : !T}\ [\text{ND}]$$

Fig. 6. Typing rules of the metalanguage

a partial function taking a list of names and a session type, and possibly returning a context:

$$x :: T \;\coloneqq\; x : T \quad (\text{when } T \text{ is rooted}) \qquad\qquad \vec{x}, \vec{y} :: (T \parallel U) \;\coloneqq\; \vec{x} :: T, \vec{y} :: U.$$

and undefined otherwise. Note that for the second equation, the decomposition in $\vec{x}, \vec{y}$ of any list $\vec{z}$ is unique, given type $T \parallel U$ of z where $\vec{x}$ and $\vec{y}$ are assigned by T and U, respectively. We write $\cdot$ for the empty list of names. Note that $\cdot :: 1$. Typing judgements are simply of the form $P \triangleright \Gamma$. The rules are described in Figure 6. (As for the syntax of untyped terms, we consider the infinite derivation tree generated by these rules.)

The rule [NIL] allows us to weaken any name preserving typability; note that π_{DiLL} is *affine* and not linear: this mirrors a similar phenomenon in game semantics. The rule [PAR] forces the split of the linear part of the context in a parallel composition. The rule [RES] introduces two connected channels with dual types.

The rule [REP] is the introduction rule for !, and corresponds to setting up a server on a which spins up a new copy of P whenever a request is made. As usual in Linear Logic, since P might be duplicated, the session types in Γ must be of the form $?S$, which we indicate with the notation $?\Gamma$. The dual rule [REQ] sends a request to a server. Note that a is still usable in the continuation.

The rule [SEL] is *selection*, an output of a payload to the other party on a channel. In the continuation, the name a disappears as it has been consumed, and the new names $\vec{x}$ appear: they are used to deconstruct the continuation type T_k. We have a non-trivial list as soon as T_k is not rooted. The dual rule [BR] represents *branching*: every possible message that can be received must be handled.

The rule [ND] arises from Differential Linear Logic and corresponds to setting up a server that accepts exactly one request, and then disappears. Unlike rule [REP], the continuation might use a again, and set up a new server later on. Two concurrent requests made on these one-time servers will race to be accepted, creating nondeterminism. This rule merges codereliction and cocontraction of DiLL in one single rule. We found easier to give non-angelic semantics to this presentation as semantics of DiLL is usually angelic. However, the usual codereliction and cocontraction are admissible from this rule.

When writing down processes, we often omit the list of successors $\vec{x}$. This may mean two things: if the move is maximal, it has no successors and we omit the empty list ϵ; if the move has a unique successor, we use the standard convention of session types which reuses the same session name, eg. we write $a \oplus k$ for $a \oplus k[a]$. This is only valid if a does not have type $?S$ since it can then be reused. In general we also reuse linear names along a session.

The forwarding agent. We introduce the *forwarding agent* $[\vec{a} \leftrightarrow \vec{b}]_S \triangleright \vec{a} :: S, \vec{b} :: S^\perp$ for any session type S and $\vec{a} :: S$ and $\vec{b} :: S^\perp$ defined. The forwarding agent represents copycat in game semantics. It is defined by induction on S, by following the η-expansion laws in Linear Logic:

$$[a \leftrightarrow b]_{\&_{i \in I} \ell_i \cdot S_i} \;=\; a \,\&\, \{\ell_i(\vec{x}).\, b \oplus \ell_i[\vec{y}].\, [\vec{x} \leftrightarrow \vec{y}]_{S_i}\}_{i \in I} \qquad [a \leftrightarrow b]_{!S} \;=\; !a(\vec{x}).\, ?a[\vec{y}].\, [\vec{x} \leftrightarrow \vec{y}]_S$$
$$[\vec{a} \leftrightarrow \vec{b}]_{S_1 \parallel \ldots \parallel S_n} \;=\; \|_{1 \le i \le n}\, [\vec{a}_i \leftrightarrow \vec{b}_i]_{S_i} \qquad [a \leftrightarrow b]_{S^\perp} \;=\; [b \leftrightarrow a]_S$$

Composition of processes. A key operation in game semantics is **composition**. It allows defining the interpretation of a programming language by introducing small bricks that can be composed together. This composition can be expressed in this setting as is well-known:

$$P \overset{\vec{a}}{\odot} Q := (\nu \vec{a}\overline{\vec{a}})(P \parallel Q) \triangleright \Gamma_1, \Gamma_2, ?\Gamma \quad \text{where} \quad P \triangleright \Gamma_1, ?\Gamma, \vec{a} : \vec{S} \text{ and } Q \triangleright \Gamma_2, ?\Gamma, \overline{\vec{a}} : \vec{S}^{\perp}$$

This mimimcks the traditional definition using parallel interaction with hiding. We use here the notation $\overline{a}$ but not with its standard π-calculus meaning: here a and $\overline{a}$ are simply two independent names which only become related via the restriction.

3.3 Operational semantics of the metalanguage

We define the operational semantics of π_{DiLL}. As for ML$_{\parallel}$, it will be done in two steps: (1) a congruence that will contain the usual structural congruence of the π-calculus, the permutation rules due to courtesy in game semantics, and finally the deterministic communication, and (2) an operational semantics which resolves the races induced by the codereliction/dereliction pairs.

Congruence. We start with the congruence on our calculus, which takes into account the *asynchronous permutations*, needed for the game semantics. These permutations are necessary to obtain a category, which is essential to interpret functional languages. These asynchronous permutations include all permutations between actions on distinct channels of the same polarity, among others. To formalise them, we use the notion of **prefix**. A prefix $\mathfrak{a}$ is a pair of an arity and a context with as many holes as the arity, as defined by:

$$\mathfrak{a} ::= a \oplus k[\vec{x}].\,[] \mid ?a[\vec{x}].\,[] \mid\ a \,\&\, \{\ell_i(\vec{x}_i).\,[]_i\}_{i\in I} \mid\ !a(\vec{x}).\,[] \mid \#a(\vec{x}).\,[]$$

The arity of all prefixes is one, except the branching prefix whose arity is the indexing set I. Given a prefix $\mathfrak{a}$ with arity I, and a family of process $(P_i)_{i\in I}$, we write $\mathfrak{a}[P_i]_{i\in I}$ for the substitution. A prefix is positive (written $\mathfrak{a}^+$) when it is a selection or dereliction; and negative otherwise (written $\mathfrak{a}^-$).

We also adopt the convention that a prefix $\mathfrak{a}^p$ is a prefix of polarity p on channel a. We write $\mathrm{bv}(\mathfrak{a})$ for the set of bound variables occurring in the pattern in $\mathfrak{a}$. Two prefixes $\mathfrak{a}$ and $\mathfrak{b}$ are orthogonal, written $\mathfrak{a} \perp \mathfrak{b}$ when $(\{a\} \cup \mathrm{bv}(\mathfrak{a})) \cap (\{b\} \cup \mathrm{bv}(\mathfrak{b})) = \emptyset$. The structural congruence on finite processes is defined as the smallest congruence on well-typed terms (over the same context) satisfying the rules described in Figure 7. The first box defines the standard structural congruence rules. The second box explains how prefixes commute with each other and other construct. The premise of [SWAP] ensures the typability of both sides of processes. The last rule [ID] is very important and states that the asynchronous forwarder behaves as an identity. This induces most asynchronous permutations. The fourth box is about deterministic communication. We consider them as a part of the congruence rules as we perform them under context – since they are deterministic, it does not matter when or where they are performed.

These rules are extended to infinite processes in the obvious continuous way: $P \equiv Q$ when for all finite $P_0 \leq P$, then there exists $Q_0 \leq Q$ with $P_0 \equiv Q_0$ and vice-versa.

Example 3.1. Remember the process P of Figure 1 representing two parallel calls to the same external function. This term is equivalent to the following:

$$P \equiv o\&\mathtt{Call}(\lambda)(f, r).\,?f[s].\,s \oplus \mathtt{Call}(1).\,?f[s'].\,s' \oplus \mathtt{Call}(2).\,s\&\mathtt{Ret}(n).\,s'\&\mathtt{Ret}(m).\,r \oplus \mathtt{Ret}(n+m).$$

This illustrates that, due to those permutations, processes that seem sequential (ie. without $\parallel$) can actually be concurrent due to these asynchronous permutations: only syntactic dependence from input to output is preserved.

$\|$ and restriction

$$P \parallel \mathbf{0} \equiv P \quad P \parallel Q \equiv Q \parallel P \quad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \quad (\nu ab)\mathbf{0} \equiv \mathbf{0} \quad (\nu ab)P \equiv (\nu ba)P$$

$$\frac{\{a,b\} \cap \{c,d\} = \emptyset}{(\nu ab)(\nu cd)P \equiv (\nu cd)(\nu ab)P} \qquad \frac{\mathrm{fv}(Q) \cap \{a,b\} = \emptyset}{(\nu ab)(P \parallel Q) \equiv ((\nu ab)P) \parallel Q}$$

Permutations of prefixes

$$\frac{}{(\nu ab)\mathfrak{a}^-[P_i]_{i\in I} \equiv \mathbf{0}}\,[\textsc{nil}] \qquad \frac{c \notin \{a,b\}}{(\nu ab)\mathfrak{c}[P_i]_{i\in I} \equiv \mathfrak{c}[(\nu ab)P_i]_{i\in i}}\,[\textsc{res}]$$

$$\frac{c \notin \{a,b\}}{(\nu ab)(\mathfrak{a}^-[P_i]_{i\in I} \parallel \mathfrak{c}^-[Q_j]_{j\in J}) \equiv \mathfrak{c}^-[(\nu ab)(\mathfrak{a}^-[P_i]_{i\in I} \parallel Q_j)]_{j\in J}}\,[\textsc{swap}] \qquad \frac{P \triangleright \Gamma, a : S}{P \overset{a}{\odot} [\tilde{a}{\leftrightarrow}b]_S \equiv P[b/a]}\,[\textsc{id}]$$

Deterministic communication

$$\frac{k \in I}{(\nu ab)(a \oplus k[\vec{x}].\, P \parallel b \,\&\, \{i(\vec{y}_i).\, Q_i\}_{i\in I}) \equiv (\nu \vec{x}\vec{y}_k)(P \parallel Q_k)}\,[\textsc{com}]$$

$$\frac{}{(\nu ab)(?a[\vec{x}].\, P \parallel !b(\vec{y}).\, Q) \equiv (\nu ab)(!b(\vec{y}).\, Q \parallel (\nu \vec{x}\vec{y})(P \parallel Q))}\,[\textsc{rep}]$$

$$\frac{}{(\nu ab)(P \parallel Q \parallel !b(\vec{x}).\, R) \equiv (\nu ab)(P \parallel !b(\vec{x}).\, R) \parallel (\nu ab)(Q \parallel !b(\vec{x}).\, R)}\,[\textsc{share}]$$

Reduction rules

$$\frac{}{(\nu ab)(?a[\vec{x}].\, P \parallel \#b(\vec{y}).\, Q \parallel R) \longrightarrow (\nu ab)(R \parallel (\nu \vec{x}\vec{y})(P \parallel Q))}\,[\textsc{race}]$$

$$\frac{P \longrightarrow Q}{E[P] \longrightarrow E[Q]}\,[\textsc{cxt}] \qquad \frac{P' \equiv P \longrightarrow Q \equiv Q'}{P \longrightarrow Q}\,[\textsc{str}]$$

Fig. 7. The structural congruence rules and reductions for π_{DiLL}

Reduction rules of π_{DiLL}. We define the operational semantics of π_{DiLL}, which now only reduces redexes involving a dereliction and a codereliction. This redex is nondeterministic because the reduction consumes the codereliction: hence two concurrent derelictions race for the codereliction. The rules of the LTS for finite processes are defined in Figure 7, where $E ::= [\,] \mid E \parallel P \mid (\nu xy)E$. As before, we extend the LTS to infinite processes by letting $P \longrightarrow Q$ when Q is the limit of all $Q_0 \leq Q$ such that there exists $P_0 \leq P$ with $P_0 \longrightarrow Q_0$. For the proof of Lemma 3.2, see Appendix A.1.

Lemma 3.2 (Subject Reduction). *If $P \triangleright \Gamma$ and $P \longrightarrow Q$, then $Q \triangleright \Gamma$.*

Observational theory. From this operational semantics, we deduce a notion of behaviour equivalence through the standard concept of *reduction-closed barbed congruence* [Honda and Yoshida 1995; Milner and Sangiorgi 1992]. First, let us define the notion of barbs: a process P has a barb on a, written $P \Downarrow a$ if $P \equiv (\nu \vec{x}\vec{y})(\mathfrak{a}^+[P] \parallel Q)$.

Definition 3.3. A reduction-closed barbed congruence is an equivalence relation on typed processes $\mathcal{R}$ containing $\equiv$ such that:

- If $(P,Q) \in \mathcal{R}$, then $P \Downarrow a$ iff $Q \Downarrow a$
- If $(P,Q) \in \mathcal{R}$ and $P \longrightarrow P'$ then there exists $(P',Q') \in \mathcal{R}$ such that $Q \longrightarrow^* Q'$.
- If $(P,Q) \in \mathcal{R}$ then for all context C, $(C[P], C[Q]) \in \mathcal{R}$.

We write $\cong_{\pi_{\mathsf{DiLL}}}$ for the largest reduction-closed barbed congurence.

4 TRANSLATION OF ML$_\parallel$ INTO π_{DiLL}

This section defines the translation of ML$_\parallel$ into π_{DiLL}, and proves it correct.

4.1 Translation of types

We start by translating types of ML$_\parallel$ into session types. As introduced in § 2.2, labels used for communication in translated types are based on the constructors of types:

$$\text{constructor:}\quad c ::= n \in \mathbb{N} \mid () \mid \mathtt{tt} \mid \mathtt{ff} \mid \lambda \mid \mathtt{ref} \qquad\qquad \text{label:}\quad \ell ::= \mathtt{Call}(c) \mid \mathtt{Ret}(c)$$

We often abbreviate $\mathtt{Call}(())$ and $\mathtt{Ret}(())$ as $\mathtt{Call}()$ and $\mathtt{Ret}()$. A type σ of ML$_\parallel$ will be interpreted as a positive session type of the form $\oplus_{c\in\mathfrak{C}(\sigma)}\mathtt{Ret}(c)\cdot(\!|\sigma|\!)_c$. Positive types are adequate to model call-by-value reduction as the initial positive move is used to tell *Context* that the term does converge to a value. The session type $(\!|\sigma|\!)_c$ represents what happens *after* the initial constructor has been sent. For base types, the constructor describes already the value, hence there is nothing to be done (ie. $(\!|\sigma|\!)_c = \mathbf{1}$). For non base types, this is more subtle. Given two positive session types we define their arrow as follows:

$$(\oplus_{i\in I}\mathtt{Ret}(\ell_i)\cdot S_i) \to T := \&_{i\in I}\mathtt{Call}(\ell_i)\cdot(S_i^\perp \parallel T).$$

The arrow type between positive types starts with a call from context with the return values of S, and then continues along $S_i^\perp$ and T. In particular T only starts after the call message has been received. Note that this construction does not preserve positivity. With these elements in hand, we can now define $(\!|\sigma|\!)$ and $(\!|\sigma|\!)_c$ by mutual induction:

$$\begin{array}{rclcrcl}(\!|\sigma|\!) & = & \oplus_{c\in\mathfrak{C}(\sigma)}\mathtt{Ret}(c)\cdot(\!|\sigma|\!)_c & \quad & (\!|\sigma\to\tau|\!)_\lambda & = & !([\![\sigma]\!]\to[\![\tau]\!]) \\ (\!|\sigma|\!)_v & = & \mathbf{1} \quad (\sigma \text{ base type}) & & (\!|\mathtt{ref}|\!)_{\mathtt{ref}} & = & !(\mathtt{get}\cdot(\!|\mathsf{int}|\!) \,\&\, \&_{n\in\mathbb{N}}\mathtt{set}(n)\cdot(\!|\bullet|\!))\end{array}$$

Reference types are interpreted as objects that can receive two methods calls: $\mathtt{get}$ and $\mathtt{set}(k)$ and that return the appropriate type. The interpretation of a context Δ is $(\!|\Delta|\!) := \oplus_{c\in\mathfrak{C}(\Delta)}\mathtt{Ret}(c)\cdot(\!|\Delta|\!)_c$ where $(\!|\Delta|\!)_c := \|_{a\in\mathsf{dom}(\Delta)}(\!|\Delta(a)|\!)_{c(a)}$: for a context, all the constructors of all the parameters arrive bundled together. The interpretaton of an interface is $(\!|\Delta\vdash\sigma|\!) := (\!|\Delta|\!)\to(\!|\sigma|\!)$ (a negative type).

4.2 Translation of terms

We want to translate a term $\Delta\vdash M:\sigma$ into a process $(\!|M|\!)_o \triangleright o : (\!|\Delta\vdash\sigma|\!)$. Doing it directly by induction would lead an inefficient translation filled with cuts that can be eliminated, as $(\!|M|\!)_o$ will always be of the shape $o\,\&\,\{c(\vec{x}).S_c\}_{c\in\mathfrak{C}(\Delta)}$ (due to the shape of $(\!|\Delta\vdash\sigma|\!)$). So the primitive object for the translation of $\Delta\vdash M:\sigma$ is what comes after receiving $c\in\mathfrak{C}(\Delta)$, ie. a family of processes $((\!|\Delta\vdash M:\sigma|\!)_{c,o} \triangleright (\!|\Delta\vdash\sigma|\!)^{\mathsf{ctx}}_{c,o})$ where $c\in\mathfrak{C}(\Delta)$, o is a name not present in Δ, and:

$$(\!|a_1:\sigma_1,\dots,a_n:\sigma_n\vdash\sigma|\!)_{c,o} = a_1:(\!|\sigma_1|\!)^\perp_{c(a_1)},\dots,a_1:(\!|\sigma_n|\!)^\perp_{c(a_n)}, o:(\!|\sigma|\!).$$

This is well-defined because for any type σ and compatible c, both $(\!|\sigma|\!)$ and $(\!|\sigma|\!)_c$ are rooted session types. From this, we can recover the usual strategy as follows:

$$(\!|a_1:\sigma_1,\dots,a_n:\sigma_n\vdash M:\sigma|\!)_o := o\,\&\,\{\mathtt{Call}(c)(a_1,\dots,a_n,o).(\!|\Delta\vdash M:\sigma|\!)_{c,o}\}_{c\in\mathfrak{C}(\Delta)}.$$

The interpretation is mostly straightforward and follows the usual game semantics intuition. To define the behaviour of $\mathtt{ref}(\cdot)$, we need to implement a reference server, ie. a server that waits for requests (get or set) and answers appropriately, while maintaining as internal state the current value of the reference. This is done using our codereliction construct $\#a(x)$. The server accepts requests one by one, and as such is infinite. We write $\mathrm{Proc}(\Gamma)$ for the ω-CPO of well-typed processes

$$(\!|\Delta, a : \sigma \vdash a : \sigma|\!)_{c,o} = o \oplus \mathtt{Ret}(c(a))[x].\, [a \leftrightarrow x]_{(\!|\sigma|\!)_{c(a)}} \qquad (\!|\Delta \vdash \underline{n} : \mathsf{int}|\!)_{c,o} = o \oplus \mathtt{Ret}(n)$$

$$(\!|\Delta \vdash b : \mathsf{bool}|\!)_{c,o} = o \oplus \mathtt{Ret}(b) \qquad (\!|\Delta \vdash () : \bullet|\!)_{c,o} = o \oplus \mathtt{Ret}()$$

$$(\!|\Delta \vdash \lambda a.\, M : \sigma \to \tau|\!)_{c,o} = \lambda^{\sigma}_{a \to o}.\, ((\!|M|\!)_{c[a:=c'],o})_{c' \in \mathfrak{C}(\sigma)}$$

$$(\!|\Delta \vdash M\, N : \sigma|\!)_{c,o} = \left((\!|M|\!)_{c,\vec{x}} \parallel (\!|N|\!)_{c,\vec{y}}\right) \overset{x,y}{\odot}$$
$$y \,\&\, \mathtt{Ret}(\lambda).\, z \,\&\, \{\mathtt{Ret}(c)(\vec{w}').\, ?y[y_0].\, y_0 \oplus \mathtt{Call}(c)[\vec{w}, o'].\, ([\vec{w} \leftrightarrow \vec{w'}] \parallel [o \leftrightarrow o'])\}_c$$

$$(\!|\vdash \mathtt{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}|\!)_{(),o} = \lambda^{\mathsf{int}}_{\cdot \to o}.\, (\lambda^{\mathsf{int}}_{\cdot \to o}.\, (o \oplus \mathtt{Ret}(i+j))_{j \in \mathbb{N}})_{i \in \mathbb{N}}$$

$$(\!|\vdash \mathtt{equal} : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}|\!)_{(),o} = \lambda^{\mathsf{int}}_{\cdot \to o}.\, (\lambda^{\mathsf{int}}_{\cdot \to o}.\, (o \oplus \mathtt{Ret}(i=j))_{j \in \mathbb{N}})_{i \in \mathbb{N}}$$

$$(\!|\vdash \mathtt{get} : \mathsf{ref} \to \mathsf{int}|\!)_{(),o} = \lambda^{\mathsf{ref}}_{r \to o}.\, (?r[x].\, x \oplus \mathtt{get}.\, r \,\&\, \{\mathtt{Ret}(n).\, o \oplus \mathtt{Ret}(n)\}_{n \in \mathbb{N}})$$

$$(\!|\vdash \mathtt{set} : \mathsf{ref} \to \mathsf{int} \to \bullet|\!)_{(),o} = \lambda^{\mathsf{ref}}_{r \to o}.\, (\lambda^{\mathsf{int}}_{\cdot \to o}.\, (?r[x].\, x \oplus \mathtt{set}(i).\, r \,\&\, \mathtt{Ret}().\, o \oplus \mathtt{Ret}())_{i \in \mathbb{N}})$$

$$(\!|\vdash \mathtt{ref} : \mathsf{int} \to \mathsf{ref}|\!)_{c,o} = \lambda^{\mathsf{int}}_{\cdot \to o}.\, (o \oplus \mathsf{ref}[a].\, \mathrm{RefServer}_a(n))_{n \in \mathbb{N}}$$

Shortcuts: $\lambda^{\sigma}_{\vec{x} \to o}.\, (P_c)_{c \in \mathfrak{C}(\sigma)} \quad := \quad o \oplus \mathtt{Ret}(\lambda).\, !o(r).\, r \,\&\, \{c(\vec{x}).\, P_c\}_{c \in \mathfrak{C}(\sigma)} \triangleright o : (\!|\sigma \to \tau|\!), ?\Delta$

Fig. 8. Interpretation of ML$\parallel$ into the metalanguage

on Γ. In particular, the least element of $\mathrm{Proc}(\Gamma)$ is $\mathbf{0}$. We then define the server via a least fixpoint of a function on the ω-CPO $\mathrm{Proc}(a : (\!|\mathsf{ref}|\!)_{\mathtt{ref}}))^{\mathbb{N}}$ (ordered pointwise) as follows:

$$\mathrm{RefServer}_a ::= \mathrm{fix}\left(\lambda S.\, \lambda n.\, \#a(x).\, x \,\&\, \begin{Bmatrix} \mathtt{get}[r].\, r \oplus \mathtt{Ret}(n).\, S(n) \\ \mathtt{set}(k)[r].\, r \oplus \mathtt{Ret}().\, S(k) \end{Bmatrix}\right)$$

The interpretation of typing rules for finite ML$\parallel$ terms (ie. without fixpoint) is given in Figure 8. Recursion is simply dealt with by unfolding it. We define the n-th approximant of the fixpoint operator by induction as: $Y_0 = \lambda\varphi.\, \lambda x.\, \bot$ and $Y_{n+1} = \lambda\varphi.\, \lambda x.\, \varphi(\lambda y.\, Y_n\ \varphi\ y)\ x$. It is straightforward to see that $(\!|Y_n|\!) \leq (\!|Y_{n+1}|\!)$ in $\mathrm{Proc}((\!|\vdash ((\sigma \to \tau) \to (\sigma \to \tau)) \to (\sigma \to \tau)|\!))$. Then, for any ML$\parallel$ term M, the n-th approximant of M, written M_n is defined as M where Y is replaced by Y_n. The chain $((\!|M_n|\!))_{n \in \mathbb{N}}$ is an increasing chain, hence has a least fixpoint which is the desired translation $(\!|M|\!)$.

4.3 Translation of machines

We now define the translation of machines. As we have seen in § 2.1, the operational semantics of machines observes memory operation via labelled transitions. To model this in the world of processes, we show how to emulate these labelled transitions.

Account of visible actions. Given an alphabet Σ, we construct a session type $(\!|\Sigma|\!) = ?\oplus_{\alpha \in \Sigma} \alpha \cdot \mathsf{ok}^{-}$. We translate machines to processes with a free name $\mathfrak{I}$ of type $(\!|\Sigma|\!)$. Such processes support the emission of actions in Σ as follows: $\alpha^{\mathfrak{I}}.\, P \;:=\; ?\mathfrak{I}[x].\, x \oplus \alpha.\, x \,\&\, \mathsf{ok}.\, P$. This allows us to define a variant of the reference server that emits a visible action at each operation performed:

$$\mathrm{RefServer}^{\mathfrak{I}}_a ::= \mathrm{fix}\left(\lambda S.\, \lambda n.\, \#a(x).\, x \,\&\, \begin{Bmatrix} \mathtt{get}[r].\, \mathrm{r}\langle \ell, n\rangle^{\mathfrak{I}}.\, r \oplus \mathtt{Ret}(n).\, S(n) \\ \mathtt{set}(k)[r].\, \mathrm{w}\langle \ell, k\rangle^{\mathfrak{I}}.\, r \oplus \mathtt{Ret}().\, S(k) \end{Bmatrix}\right)$$

Unlike the previous server, this one logs (emit a visible action) everytime an action is performed. Given a state $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle$ we define the process

$$(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathfrak{I}} = (\nu o\bar{o})(\nu \vec{a}\bar{\vec{a}})\left((\!|M|\!)_{c_\Delta, o} \parallel (\|_{a \in \vec{y}}\ \mathrm{RefServer}^{\mathfrak{I}}_{\vec{a}}) \parallel (\|_{a \in \mathrm{dom}(\Delta) \setminus \vec{y}}\ \mathrm{RefServer}_{\vec{a}})\right)$$

with $\vec{a} \in \mathrm{dom}(\Delta)$. We only use the logging reference server for public names: every operation gives rise to a visible action. Note that the interpretation of a machine is a closed process, thus

there is no need for an output channel as for the terms, as the return value is discarded. Moreover, in this case, $\mathfrak{C}(\Delta)$ is a singleton, hence the parameter c_Δ is simply the tuple $(\mathsf{ref}, \ldots, \mathsf{ref})$.

Weak bisimulation and barbed reduction-closed congruence of $\pi_{\mathbf{DiLL}}$. On processes well-typed in the context $\mathrm{I} : (\!|\Sigma|\!)$, we can define a LTS as follows:

$$P \xrightarrow{\tau} Q \text{ whenever } P \to Q \qquad P \xrightarrow{\alpha} Q \text{ whenever } P \equiv \alpha^{\mathrm{I}}.\, Q.$$

Using this LTS, we can define a notion of weak bisimulation on such processes.

Definition 4.1 (Weak bisimulation of π_{DiLL}). An equivalence relation $\mathcal{R}$ over processes typed in the context $\mathrm{I} : (\!|\Sigma|\!)$, is a *weak bisimulation* if for all $P \mathcal{R} Q$ with $P, Q \triangleright \emptyset$, $P \xrightarrow{\alpha} P'$, there exists $Q \xRightarrow{\hat{\alpha}} Q'$ with $P' \mathcal{R} Q'$. The largest such bisimulation is called **weak bisimilarity**, denoted by $\approx_{\pi_{\mathrm{DiLL}}}$.

4.4 Correctness of the translation

Soundness of call-by-value. We first show that our semantics is well-behaved with respect to the parallel call-by-value strategy. First, we observe that values get mapped to specific processes – those which emit a constructor right away, without interrogating the context. Formally, given a value $\Delta \vdash V : \sigma$ and given $c \in \mathfrak{C}(\Delta)$, we define $\mathfrak{C}(c, V) \in \mathfrak{C}(\sigma)$ by induction on V:

$$\mathfrak{C}(c, x) = c(x) \qquad \mathfrak{C}(c, \underline{n}) = n \qquad \mathfrak{C}(c, b) = b \quad (b \in \{\mathsf{tt}, \mathsf{ff}, ()\}) \qquad \mathfrak{C}(c, \lambda x.\, M) = \lambda$$

Lemma 4.2. *Assume $\Delta \vdash V : \sigma$. Then $(\!|\Delta \vdash V : \sigma|\!)_{c,o} \equiv o \oplus \mathtt{Ret}(\mathfrak{C}(c, V)).\, P$ for some process P.*

Lemma 4.3. *For $\Delta \vdash V : \sigma$ and $\Delta, x : \sigma \vdash M : \tau$, $(\!|M[V/x]|\!)_{c,o} \equiv (\!|M|\!)_{c[x := \mathfrak{C}(c,V)],o} \overset{x}{\odot} !\bar{x}(r).\, (\!|V|\!)_{c,r}$.*

Lemma 4.4. *Consider $\Delta \vdash M, N : \sigma$. If $M \to N$, then for all c, o, $(\!|M|\!)_{c,o} \equiv (\!|N|\!)_{c,o}$.*

Full-abstraction on machines. We now turn our attention to the LTS on machines. We first start with a simulation lemma.

Lemma 4.5. *If $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$, then $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \xRightarrow{\alpha} (\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!)_{\mathrm{I}}$.*

We now show an adequacy lemma which shows the converse. However, formulating this result is subtle due to slight differences in the LTS. In $\mathsf{ML}_{\|}$, operations on visible locations are performed directly as visible transitions: $x \vdash^{x:\mathsf{ref}} \langle x := 1 \parallel x := 2, x \mapsto 0\rangle$ can do either $\mathrm{w}\langle x, 1\rangle$ or $\mathrm{w}\langle x, 2\rangle$ as initial transitions. In π_{DiLL}, any transition is done in two steps: first the program connects to the memory server, which implies that a dereliction meets a coderelication; and only then the action is logged (cf. the definition of $\mathtt{RefServer}^{\ell}$). This means that the translation of this machine can do two τ-transitions, each followed by a write action. Hence, the two LTSs are not directly weakly bisimilar, but we can still show the following result:

Lemma 4.6 (Adequacy). *If $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \xrightarrow{\alpha} P$, then $\alpha = \tau$, and there are two cases:*

- *Either $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\tau} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ with $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!)_{\mathrm{I}} \equiv P$,*
- *Or $P \equiv \alpha^{\mathrm{I}}.\, Q$ and $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ with $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!)_{\mathrm{I}} \equiv Q$.*

From this, we can show that our translation preserves and reflects weak bisimulation:

Theorem 4.7 (Soundness and completeness). *For any typed $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle$ and $\Delta' \vdash^{\vec{y}} \langle N, \mu'\rangle$, the following are equivalent:* (1) $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \approx_{\mathsf{ML}_{\|}} \Delta \vdash^{\vec{y}} \langle N, \mu'\rangle$, (2) $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \approx_{\pi_{DiLL}} (\!|\Delta \vdash^{\vec{y}} \langle N, \mu'\rangle|\!)_{\mathrm{I}}$ *and* (3) $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \cong_{\pi_{DiLL}} (\!|\Delta \vdash^{\vec{y}} \langle N, \mu'\rangle|\!)_{\mathrm{I}}$

From Theorem 4.7, it follows immediately that:

Theorem 4.8 (Soundness). *For all* $\Delta \vdash M, N : \sigma$, *if* $(\!|N|\!)_o \cong_{\pi_{DiLL}} (\!|N|\!)_o$ *then* $M \cong_{ML_{\|}} N$.

The converse, known as *full abstraction* in the denotational semantics community does not hold in general for very well-understood reasons. See the Appendix § B.4 for a counterexample. However, we still have full abstraction at second-order interfaces, cf. § 6.3.

5 CAUSAL GAME SEMANTICS

In this section, we describe a causal semantics interpretation of π_{DiLL}. This model interprets session types and processes as event structures, a causal model of concurrent and nondeterministic computation. The event structures denoted by session types are called **games**, while the event structures denoted by processes are called **strategies**. The contribution of this section is to extend the non-angelic model of [Castellan et al. 2018] with the tools for replication of [Castellan et al. 2019]. This results in the first game semantics model providing adequate modelling of nondeterministic and nonlinear languages up to weak bisimulation.

In § 5.1, we recall Winskel's event structures and introduce the notations we will be using. In § 5.2, we define the notion of games of [Rideau and Winskel 2011], as well as the interpretation of type formers on it. In § 5.3, we present strategies based on event structures of [Castellan et al. 2018]. In § 5.4, we introduce composition of strategies as well as the categorical structure which is key to interpret π_{DiLL}. Finally, in § 5.5, we show problems related to replication and introduce ***symmetry*** to solve them.

5.1 Event Structures

Our model is based on event structures [Winskel 1986], introduced to give a semantic model of causal concurrency (or true concurrency). From this viewpoint, a system is a set of events along with a partial order describing the **causal relationship** between events; and a binary relation describing **conflict** (or incompatibility) between events, ie. those events that cannot occur together in the same run of the system. As a result, event structures model the system **globally** rather than describe independent executions. We use here *prime event structures with binary conflict.*

Definition 5.1. An **event structure** is a triple $(E, \leq_E, \#)$ where $(E, \leq_E)$ is a partial order, and $\# \subseteq E^2$ is a binary irreflexive symmetric relation satisfying: (1) $[e] \stackrel{\text{def}}{=} \{e' \in E \mid e' \leq e\}$ is finite, and (2) if $e \# e'$ and $e' \leq e''$ then $e \# e''$.

Two events e, e' are in **conflict** when $e \# e'$ and are **compatible** otherwise. Two compatible events which are not ordered are called **concurrent**. We write $e \rightarrowtriangle e'$ (**immediate causal dependency**) when $e <_E e'$ with no events in between, and $e \sim e'$ (**minimal conflict**) when (e, e') is the only conflicting pair in $[e] \cup [e']$. From $\rightarrowtriangle$ and $\sim$, we can recover $\leq$ and $\#$ via axiom (2). Depictions of event structures will use $\rightarrowtriangle$ and $\sim$.

A notion of partial execution can be recoverd on event structures through the notion of configuration. Given an event structure E, a **configuration** of E is a subset $x \subseteq E$ down-closed for $\leq_E$ and conflict-free. We write $\mathcal{C}(E)$ for the set of finite configurations of E. For $x \in \mathcal{C}(E)$, an **extension** of x is an event $e \in E \setminus x$ such that $x \cup \{e\} \in \mathcal{C}(E)$, which we write $x \overset{e}{\text{—}\subset}$.

An event structure E is **confusion-free** when (1) $e \sim e'$ implies $[e] \setminus \{e\} = [e'] \setminus \{e'\}$ and (2) the relation $(\sim_E \cup =_E)$ is an equivalence relation. Its equivalence classes are called **cells**. Confusion-free event structure have **local** nondeterminism and cells represent choices between different branches of the program. Finally, given a set $V \subseteq E$, the **projection** of E to V is the event structure $E \downarrow V \stackrel{\text{def}}{=} (V, \leq_E \cap V^2, \# \cap V^2)$.

Constructions on Event Structures. Given a family of event structures $(E_{i \in I})$ we define their **parallel composition** $\|_{i \in I} E_i$ as follows. Its events are pairs $(i \in I, e \in E_i)$. Causality and conflict are

obtained by lifting those from the E_i:

$$(i,e) \leq_{\|_{E_i}} (j,e') \stackrel{\text{def}}{=} (i = j \wedge e \leq_{A_i} e') \qquad (i,e)\#_{\|_{E_i}}(j,e') \stackrel{\text{def}}{=} (i = j \wedge e\#_{A_i}e')$$

Similarly, the **nondeterministic sum** of the E_i, written $\sum_{i\in I} E_i$ has the same components as $\|_{i\in I}$ E_i except for conflict:

$$(i,e)\#_{\sum_{i\in I} E_i}(j,e') \stackrel{\text{def}}{=} (i = j \Rightarrow e\#_{E_i}e').$$

The empty event structure, written $\emptyset$ is the unit for both parallel composition and sum.

5.2 Games based on Event Structures

We use event structures to represent the semantic information of session types. This is a natural fit as session types feature both **causality**, for some messages must come before some other, and **conflict**, for choices are offered to both participant. In this paper, we use a simple notion of games.

Definition 5.2. A **game** is an event structure A along with a labelling $pol : A \to \{-,+\}$ such that:

(1) A is confusion-free and **race-free**: if $a \sim_A a'$, then $pol(a) = pol(a')$
(2) $(A, \leq_A)$ is a **forest**: elements of $[a]$ are totally ordered by $\leq_A$ for any $a \in A$.

The first axiom comes from the local and polarised aspects of choices in session types of π_{DiLL}. Parallel composition of event structures extends to games. Given a game A and $\ell \notin A$, we write $\ell^p \cdot A$, where $p \in \{-,+\}$ is a polarity, for the game A prefixed by ℓ^p defined as follows. Its events are $A \cup \{\ell\}$; causality is $\leq_A \cup \{\ell\} \times (A \cup \{\ell\})$; and conflict is $\#_A$. In general sums of games are not games as they may fail confusion-freeness or race-freeness, but the prefixed sum of games is well-defined: if A_i is a family of games, then $\sum_{i\in I} \ell_i^p \cdot A_i$ is also a game (because the polarity p does not depend on on i). Given a game A, its **dual** $A^\perp$ is the game obtained by reversing polarities in A.

Moves of a game are meant to be played once, therefore to represent exponentials we need to make copies of a game to allow a move to be played several times. We define the unpolarised exponential $\sharp A$ as the infinite parallel composition $\|_{i\in\mathbb{N}} A$. Events of $\sharp A$ are pairs (i,a) of a **copy index** $i \in \mathbb{N}$ and an element $a \in A$. We sometimes write a_i (especially in diagrams).

Examples of games related to the interpretation of $\mathsf{ML}_\|$ are found in Figures 4 and 5.

5.3 Strategies as Event Structures

We recall the strategies of [Castellan et al. 2018]. As we have seen in § 2, the game expresses all possible behaviours allowed by the type, while the strategy selects those behaviours exhibited by the program. As a result, in a deterministic setting, a strategy on a game A is actually a subset of A. In general though, the strategy might make nonidempotent choices, so a strategy will be an event structure S, along with a labelling function $S \to A$. For the strategy to respect the rules of the game (eg. if $a < b$, then always play a before b), we need the function to be a *map of event structures*, ensuring that every behaviour of S is included in that of A.

Definition 5.3. A **partial map of event structures** is a partial function $f : E \rightharpoonup F$ such that for $x \in \mathcal{C}(E)$, $fx \in \mathcal{C}(F)$ and which is **locally injective**: f restricted to any configuration of E is injective. We write $\mathrm{dom}(f)$ for the domain of f, and f is said to be **total** when $\mathrm{dom}(f) = E$. A map represents a simulation of E by F. Event structures and total maps form a category **ES**.

Definition 5.4. A **strategy** on a game A is a pair (S,σ) of an event structure S and a partial map of event structures $\sigma : S \rightharpoonup A$ with the following properties:

Receptivity For $x \in \mathcal{C}(S)$ and a a negative extension of $\sigma\, x$, there is a unique $x \stackrel{s}{\text{—}\!\subset} $ with $\sigma\, s = a$.

Courtesy For any $s \rightarrow_S s'$ such that $\sigma\, s$ and $\sigma\, s'$ are not related by $\leq_A$ (possibly because one of them is undefined), the event s is not mapped to a positive move of A, and s' not mapped to a negative move of A.

Secrecy If $s \sim s'$, then s and s' are not mapped to positive moves of A.

Partiality allows S to have internal events (akin to τ transitions) not corresponding to any moves of the game. An event $s \in S$ is **internal** or neutral when $\sigma\, s$ is not defined; **visible** or external when it is. Receptivity ensures that a strategy cannot prevent Opponent from playing. Courtesy restricts the shape of immediate causal links in the strategy to only be from negative or neutral events to positive or neutral. Those links are asynchronous and are essential to ensure that we have a categorical identity. Finally, secrecy (together with receptivity) restricts the shape of minimal conflict: either between negative moves or between internal moves. The former are external choices (Opponent chooses) while the latter are internal choices (Player chooses).

We tend to write simply $\sigma, \tau, \ldots$ for strategies and assume the underlying event structure is called $S, T, \ldots$ A strategy from a game A to a game B is a strategy on the composite game $A^\perp \parallel B$. We write $\sigma : A \nrightarrow B$ to distinguish them from maps. Strategies are displayed as labelled event structures, as in Figures 4 and 5, which are examples of total strategies, ie. for which there are no internal moves. Because of the conflict axiom, such strategies must be also deterministic (no internal choices, only external choices). On the right, the interpretation as a strategy of the nondeterministic program $(x := 1 \parallel x := 2); 1$ is depicted. The nondeterminism arises from a race between two writes on the same location. In this example, the map σ is not injective (it plays twice $\mathtt{Ret}(1)$), because there might be two reasons for returning this value, depending on how the race is resolved. In the diagram, we use $*$ for neutral events, annotated with a label describing the operation —but this information is not part of the mathematical object.

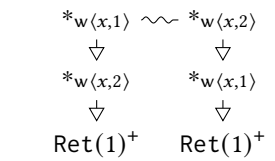


Fig. 9. Nondeterministic strategy

Copycat strategy. For a game A, we form the **copycat strategy** on A, $\mathfrak{cc}_A : \mathbb{C}_A \to A^\perp \parallel A$, as follows. The events of $\mathbb{C}_A$ are exactly those of $A^\perp \parallel A$, and $\leq_{\mathbb{C}_A}$ is obtained from the transitive closure of $\leq_{A^\perp \parallel A} \cup \{(a, \bar{a}) \mid a \in A^\perp \parallel A \text{ is negative}\}$ where we write $\bar{a} \in A^\perp \parallel A$ for the corresponding event to a on the other side. Then $a \#_{\mathbb{C}_A} a'$ holds when $[a]_{\mathbb{C}_A} \cup [a']_{\mathbb{C}_A} \notin C(A^\perp \parallel A)$. The labelling function $\mathfrak{cc}_A$ is simply the identity. A strategy is **negative** when its minimal events are all negative. Copycat for example is a negative strategy.

The empty strategy $!_A$ on a game A is defined as the inclusion $A_0 \subseteq A$ where A_0 contains events $a \in A$ such that $[a]$ only contains negative events (ie. it is the minimal receptive strategy).

Isomorphism of strategies. Because S is arbitrary, equality of strategies is not meaningful. Rather, we consider strategies up to isomorphism. Two strategies $\sigma : S \rightharpoonup A$ and $\tau : T \rightharpoonup A$ are **isomorphic** (written $\sigma \cong \tau$) when there exists an isomorphism of event structures $\varphi : S \cong T$ (that is a map $S \to T$ with an inverse map) such that $\tau \circ \varphi = \sigma$.

5.4 Composition of strategies and categorical structure

We now explore an important algebraic structure on strategies: composition. Composition is the semantic counterpart to the operation $\overset{a}{\odot}$ which connects two strategies and make them interact on a common part. Concretely, given $\sigma : A \nrightarrow B$ and $\tau : B \nrightarrow C$, we wish to define $\tau \odot \sigma : A \nrightarrow C$ obtained by making σ and τ interact on B and hiding the resulting interaction. This composition allows us to define a category of games and strategies. This is useful for two main reasons:

- It allows us to give a simple meaning to the restriction $(vab)P$ by simply composing P with an asynchronous forwarder that connects a and b.
- It allows us to define the interpretation of constructs of the metalanguage by composing with simple blocks rather than some ad-hoc construction. For instance, to interpret $a \oplus \ell.\, P$, instead of somehow prefixing $[\![P]\!]$ by an event labelled with $a \oplus \ell$, we prefer to compose $[\![P]\!]$ by a strategy $\iota_{a\oplus\ell}$ which represents an injection for a weak coproduct structure. This allows us to show the correctness of the translation by simple calculations on these little blocks rather than with an arbitrary context P.

Composition of Strategies. Composition is defined in two steps: interaction, and then hiding. Consider a strategy $\sigma : A \nrightarrow B$ from A to B, and a strategy $\tau : B \nrightarrow C$ from B to C. We first start by forming their interaction, which is an event structure $T \circledast S$ along with a map $\tau \circledast \sigma : T \circledast S \rightharpoonup A^\perp \parallel B \parallel C$ (during interaction, polarities on B are meaningless). This event structure represents the common behaviours of S and T that can be reached through a common order. Intuitively, on the causal aspect, this amounts to make the union of the partial orders of S and T and remove cycles. This operation is somewhat involved to define concretely, but enjoys the following universal property.[3]

Lemma 5.5 ([Castellan et al. 2018]). *There exists a unique tuple* $(T \circledast S, \tau \circledast \sigma, \Pi_1 : T \circledast S \rightharpoonup S, \Pi_2 : T \circledast S \rightharpoonup T)$ *such that*

(1) If $\Pi_1\, e$ *and* $\Pi_2\, e$ *are defined then* $\sigma(\Pi_1\, e)$ *and* $\tau(\Pi_2\, e)$ *are both defined and equal (in* B*).*
(2) Moreover, for any other tuple $(X, \chi : X \rightharpoonup A \parallel B \parallel C, \Pi_1' : X \rightharpoonup S, \Pi_2' : X \rightharpoonup T)$ *satisfying the first property, there exists a unique map* $\varphi : X \rightharpoonup T \circledast S$ *such that* $\Pi_1 \circ \varphi = \Pi_1'$ *and* $\Pi_2 \circ \varphi = \Pi_2'$.

This ensures that $T \circledast S$ is well-defined up to isomorphism. The second step is to recover a strategy on $A^\perp \parallel C$ by hiding those events of $T \circledast S$ projected to B. Because of secrecy, events in B cannot partake in a minimal conflict, so they can be hidden without losing information up to bisimulation.

Letting V be the set of events e of $T \circledast S$ such that if $\tau \circledast \sigma(e)$ is defined, it is not in B, we define $T \odot S$ to be $T \circledast S \downarrow V$. The map $\tau \circledast \sigma$ restricts to a map $\tau \odot \sigma : T \odot S \rightharpoonup A^\perp \parallel C$ which is the **composition** of σ and τ. It is a strategy.

Lemma 5.6 ([Castellan et al. 2018]). *Composition of strategies is associative, and copycat is an identity on strategies, both up to isomorphism.*

For copycat to be an identity, the conditions of receptivity and courtesy are necessary. This is because composing a strategy with copycat amounts to prefixing it with an asynchronous buffer; and if the strategy is too "synchronous", this buffer may alter its observable behaviour.

Categorical structure. We define **Str**, the category whose objects are finite games and morphisms from A to B are strategies $\sigma : A \nrightarrow B$, considered up to isomorphism. First, duality of games extends to an isomorphism of category $\mathbf{Str} \cong \mathbf{Str}^{\mathrm{op}}$: any strategy $\sigma : A \nrightarrow B$ can be regarded as a strategy $\sigma^\perp : B^\perp \nrightarrow A^\perp$. Moreover, parallel composition of games forms a monoidal structure on **Str**. Therefore, **Str** is compact-closed:

Lemma 5.7 ([Castellan et al. 2018]). $(\mathbf{Str}, \parallel, \mathbf{1}, (-)^\perp)$ *is a compact-closed category.*

Though **Str** does not have (co)products, it has weak (co)products. Given a family of games $(A_k)_{k\in K}$ and of labels $(\ell_k)_{k\in K}$, we define $\&_{k\in K} \ell_k^- \cdot A_k \stackrel{\text{def}}{=} \sum_{k\in K} \ell_k^- \cdot A_k$. There is a natural strategy $\pi_k : \&_{k\in K} \ell_k^- \cdot A_k \nrightarrow A_k$ which plays ℓ_k on the right (a positive move as it is on the left), and then plays copycat between the left A_k and the right A_k.

[3] In the case without internal events, this is a pullback, but in this setting, the pullback exists but does not capture the right operational intuitions.

Lemma 5.8. *For every family of strategies* $(\sigma_k : A \nrightarrow B_k)_{k\in K}$ *there exists a strategy* $\langle\sigma_k\rangle_{k\in K} : A \nrightarrow \&_{k\in K}\, \ell_k^- \cdot B_k$ *such that* $\pi_k \odot \langle\sigma_k\rangle_{k\in K} \cong \sigma_k$

By duality, we get weak coproducts: $\oplus_{k\in K}\ell_k^+ \cdot A_k \stackrel{\text{def}}{=} \sum_{k\in K} \ell_k^+ \cdot A_k$, and define

$$\iota_k \stackrel{\text{def}}{=} \pi_k^\perp : A_k \nrightarrow \oplus_{k\in K}\, \ell_k^+ \cdot A_k.$$

These are used to interpret the additive fragment of the metalanguage.

5.5 Replication and symmetry

We have weak co-products, and compact-closedness so we can interpret a large part of π_{DiLL}. For exponentials, we would be tempted to use $\sharp(-)$ construction to interpret $!(\cdot)$ and $?(\cdot)$. However, for the interpretation be to sound, we would require special structural morphisms. It is well-known from the Linear Logic categorical semantics [Melliès 2009] that we would need $\sharp(-)$ to have a (co)monadic structure to be able to interpret correctly the promotion, and dereliction rule.

There are natural candidates for a monad structure on $\sharp(-)$:

$$\eta_E : (a \in E) \mapsto (0, a) \in \sharp E \qquad\qquad \mu_E : \big((i,(j,a)) \in \sharp\sharp E\big) \mapsto \big((\langle i, j\rangle, a) \in \sharp E\big)$$

These can be lifted to strategies, however one can see already in **ES** those do not satisfy the monadic laws: indeed $\mu_E \circ \eta_E : \sharp A \to \sharp A$ sends (i, a) to $(\langle i, 0\rangle, a)$ which may not be the same as (i, a).

Event Structures with Symmetry. The solution to this problem is to relax the notion of equality to be *up to copy indices*. This intuition is formalised through **event structures with symmetry**:

Definition 5.9. An **isomorphism family** on an event structure E is a family $\tilde{E}$ of order-isomorphisms $\varphi : x \cong y$ with $x, y \in \mathcal{C}(E)$ such that

(1) $\tilde{E}$ contains all identities, and is stable under inverse and composition
(2) If $x \subseteq x'$ and $\varphi : x' \cong y' \in \tilde{E}$, then $\varphi|_x : x \cong \varphi\, x \in \tilde{E}$
(3) If $x \subseteq x'$ and $\varphi : x \cong y \in \tilde{E}$, then there exists a (non-necessarily unique) $y' \in \mathcal{C}(E)$ such that φ extends to $\varphi' : x' \cong y' \in \tilde{E}$.

An **event structure with symmetry** is a pair $(E, \tilde{E})$ of an event structure E and an isomorphism family $\tilde{E}$ on E. We write $\mathcal{E}, \mathcal{F}, \ldots$ for event structures with symmetry.

Most constructions on event structures (such as parallel composition, sum, prefixing) can be extended seamlessly to event structures with symmetry [Winskel 2007]. The main purpose of these objects is to support a weaker equivalence relation on maps. Consider two partial maps $f, g : E \rightharpoonup F$ between event structures and an isomorphism family $\tilde{F}$ on F. These maps are **similar** wrt $\tilde{F}$, written $f \sim_{\tilde{F}} g$ when for all $x \in \mathcal{C}(E)$, the mapping $f\, e \mapsto g\, e$ defines a bijection $f\, x \cong g\, x$ which is in $\tilde{F}$. When the family is clear from context, we write $f \sim g$.

Now, we can extend the $\sharp(-)$ construction to work at the level of symmetry. Given an event structure with symmetry $\mathcal{E}$, define an isomorphism family $\sharp\tilde{E}$ on $\sharp E$ containing all the $\theta : x \cong y$ for which there exists $\pi : \mathbb{N} \to \mathbb{N}$ such that (1) for $(i, e) \in x$, $\theta\,(i, e)$ is of the form $(\pi\, i, e')$and(2) $\{(e, e') \mid \theta(i, e) = (\pi\, i, e')\} \in \tilde{E}$. We define the event structure with symmetry $\sharp\mathcal{E}$ as $(\sharp E, \sharp\tilde{E})$. The induced equivalence formally captures to be equal "up to copy indices".

A **map of event structures with symmetry** $f : \mathcal{E} \rightharpoonup \mathcal{F}$ is a map $f : E \rightharpoonup F$ such that for all $\theta \in \tilde{E}$, $f\,\theta \stackrel{\text{def}}{=} \{(f\, e, f\, e') \mid (e, e') \in \theta\}$ is in $\tilde{F}$. Event structures with symmetry and total maps form a category **ESS**. Winskel [2007] proved that $\sharp(-) : \mathbf{ESS} \to \mathbf{ESS}$ is a monad up to $\sim$.

Thin concurrent games with symmetry. Simply equipping games with one symmetry works [Castellan et al. 2014] but leads to a setting uncomfortable for semantics. We follow here [Castellan et al. 2019] and add three symmetries on games: one global symmetry, and one subsymmetry for each player. (Details of this section appear in the unpublished thesis [Castellan 2017]).

Definition 5.10. A **thin concurrent game** (tcg) is a tuple $\mathcal{A} = (A, \tilde{A}, \tilde{A}_+, \tilde{A}_-)$ where A is a game, and $\tilde{A}, \tilde{A}_+, \tilde{A}_-$ are isomorphism families on A subject to axioms listed in [Castellan et al. 2019]; in particular $\tilde{A}_-$ and $\tilde{A}_+$ are sub-isomorphism families of $\tilde{A}$.

The intuition is that $\tilde{A}_-$ (resp. $\tilde{A}_+$) contains only isomorphisms that affect negative (resp. positive) events. This idea of polarised decomposition of the symmetry was first introduced by Melliès [2003] in a simpler setting. Game operations (parallel composition, dual, prefixed sums) extend to tcgs.

However, in general $\sharp\mathcal{A}$ is not a tcg even if $\mathcal{A}$ is. It only works for polarised games: a game is **negative** (resp. **positive**), when all its minimal events are negative (resp. positive). A game is **polarised** when it is either negative or positive.

Lemma 5.11 ([Castellan et al. 2019]). *If $\mathcal{A}$ is a polarised tcg, then $\sharp\mathcal{A}$ can be made into a tcg.*

A negative symmetry in $\sharp\mathcal{A}$ leaves *positive* copy indices unchanged, and conversely for positive symmetries. Since the interpretation of session types is not polarised, we need to explicitly *lift* them before using $\sharp(-)$: indeed a lifted game is always polarised as it has a unique minimal event. Depending on the polarity of the minimal event, we get the two exponentials. We define:

$$!\mathcal{A} = \sharp(\mathsf{Req}^- \cdot \mathcal{A}) \qquad ?\mathcal{A} = \sharp(\mathsf{Req}^+ \cdot \mathcal{A}).$$

Uniform Strategies. Symmetry on games induces naturally an equivalence relation on strategies, relaxing isomorphism by asking that the triangle commutes up to symmetry on the game. This equivalence, however, is not a congruence since nothing prevents a strategy from observing copy indices from Opponent. To recover a well-behaved compositional setting, we need to ensure that the strategy we consider behaves uniformly with respect to copy indices from Opponent. It turns out that we need to add extra structures on strategies, under the form of an isomorphism family.

Definition 5.12. Consider a tcg $\mathcal{A}$. A **uniformity witness** for an essential strategy $\sigma : S \rightharpoonup A$ is an isomorphism family $\tilde{S}$ on S such that:

(1) σ becomes a map of event structures with symmetry $(S, \tilde{S}) \to (A, \tilde{A})$;
(2) if $\theta : x \cong y \in \tilde{S}$ and $\sigma\,\theta$ extends to $\varphi : x' \cong y' \in \tilde{A}$ with $x \subseteq^- x'$, then θ extends to a θ' such that $\sigma\,\theta' = \varphi$;
(3) if $\theta : x \cong y \in \tilde{S}$ is the identity on negative elements of x, then θ is the identity on x.

A **uniform strategy** on $\mathcal{A}$ is a strategy σ on A along with a uniformity witness for σ.

We can lift the equivalence relation $\sim$ from maps to strategies: two uniform strategies $\sigma : S \to A$ and $\tau : T \to A$ are **weakly isomorphic** (written $\sigma \cong \tau$) when there exists an isomorphism of event structures with symmetry $\varphi : \mathcal{S} \cong \mathcal{T}$ such that $\tau \circ \varphi \sim \sigma$.

Theorem 5.13 ([Castellan 2017]). *Uniformity witnesses compose and weak isomorphism is a congruence on uniform strategies. As result, tcgs and uniform strategies up to weak isomorphism form a compact-closed category* **UStr**.

The weak products of **Str** induce weak products in **UStr** in a straighforward manner. We also get the desired In this settting, we get the desired (co)monadic structure:

Lemma 5.14. *The construction* $!(-)$ *extends to an exponential comonad* **UStr** $\to$ **UStr**, *that is it comes equipped a functorial action plus the following strategies:*

return: $\mathbf{d}_{\mathcal{A}} : !\mathcal{A} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathcal{A}$ ***digging:*** $\mu_{\mathcal{A}} : !\mathcal{A} \rightarrowtail\!\!\!\!\twoheadrightarrow !!\mathcal{A}$ ***contraction:*** $\mathbf{c}_{\mathcal{A}} : !A \rightarrowtail\!\!\!\!\twoheadrightarrow !A \parallel !A$.

satisfying standard laws [Melliès 2009].

Recursion. We finish by detailing the structure in **UStr** to interpret infinite processes. An **embedding** of σ into τ (both strategies on $\mathcal{A}$) is a map $f : \mathcal{S} \to \mathcal{T}$ such that $\tau \circ f \sim \sigma$ and such that f restricted to its image defines an isomorphism. We write $f : \sigma \hookrightarrow \tau$. This notion is useful to define the meaning of infinite objects by taking a limit of increasing strategies.

Lemma 5.15. *Any countable chain of embeddings* $(f_i : \sigma_i \hookrightarrow \sigma_{i+1})_{i\in\mathbb{N}}$ *has a colimit, ie. there exists a unique a strategy* σ *and embeddings* $g_i : \sigma_i \hookrightarrow \sigma$ *such that* $g_{i+1} \circ f_i = g_i$.

6 INTERPRETATION OF π_{DiLL} INTO STRATEGIES

We are ready to interpret π_{DiLL} into **UStr**. In § 6.1, we show how to interpret session types and processes. In § 6.2, we prove that the interpretation is fully abstract with respect to a weak bisimulation on strategies. Finally, in § 6.3 we discuss the resulting causal semantics of $\mathsf{ML}_{\parallel}$.

6.1 Semantics interpretation

Interpretation of types. The interpretation of types is straightforward, since we have seen the semantic constructions corresponding to type formers in § 5: $[\![!T]\!] = ![\![T]\!]$, $[\![?T]\!] = ?[\![T]\!]$,

$$[\![T_1 \parallel \ldots \parallel T_n]\!] = [\![T_1]\!] \parallel \ldots \parallel [\![T_n]\!], \quad [\![\&_{i\in I}\ell_i^- \cdot T_i]\!] = \&_{i\in I}\ell_i^- \cdot [\![T_i]\!] \text{ and } [\![\oplus_{i\in I}\ell_i^+ \cdot T_i]\!] = \oplus_{i\in I}\ell_i^+ \cdot [\![T_i]\!].$$

Minimal events of $[\![S]\!]$ are either of the form $\ell \in \mathbb{L}$ (a label) or Req for starting new requests. A session type S is rooted if and only if all minimal events of $[\![S]\!]$ are in conflict. The interpretation of contexts is a simple parallel composition: $[\![a_1 : S_1, \ldots, a_n : S_n]\!] = [\![S_1]\!] \parallel \ldots \parallel [\![S_n]\!]$. Minimal events of $[\![\Delta]\!]$ are of the form (a, v) where $a \in \mathrm{dom}(\Delta)$ and v is the minimal event of $[\![\Delta(a)]\!]$.

Interpretation of terms. To interpret terms, we make use of the categorical structure presented in § 5. In particular, if $\vec{x} :: S$ is defined, then there is an isomorphism $\varphi_{\vec{x}::S} : [\![\vec{x} :: S]\!] \cong [\![S]\!]$ obtained from the monoidal structure. Given an isomorphism $\varphi : \mathcal{A} \cong \mathcal{B}$ between tcgs, and $\sigma : \mathcal{A}$ a uniform strategy, we write $\varphi^*(\sigma) : \mathcal{B}$ for the reindexed strategy on $\mathcal{B}$.

What remains to be constructed in the semantic world is how to interpret codereliction, ie. how to introduce nondeterminism and races. Such a construct is interpreted in **UStr** through a morphism $\bar{d}_A : !A \parallel A \rightarrowtail\!\!\!\!\twoheadrightarrow !A$.

The intuition is that requests on the right side race to get forwarded to A; requests that did not win the race are then forwarded to the left $!A$. Since this strategy has a complex operational behaviour, let us start by an informal description, based on the example where the game has a single positive move, ie. $A = a^+$. The desired strategy $\bar{d}_A$ for a single move game $A = \mathsf{a}^+$ is (partially) depicted in Figure 10. The strategy starts by waiting for a request on the rightmost $!A$. The different Req_i^- are all racing together to be considered the **first** message acknowledged by $\bar{d}_A$. Hence, one neutral event per Req_i^- is triggered when the ith message wins the race. If the ith message has won, then the copy of A on the left is put in contact with the successor of Req_i^-, expressed by the links $a^- \twoheadrightarrow \langle a, i\rangle^+$ and $*_i \twoheadrightarrow \langle a, i\rangle^+$.

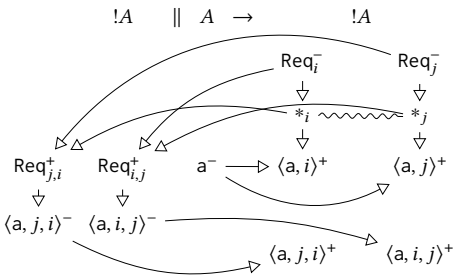


Fig. 10. Strategy $\bar{d}_A$ for $A = \mathsf{a}^+$

$$\left[\!\left[\frac{}{\mathbf{0} \triangleright \Delta}\right]\!\right] = !_\Delta \qquad \left[\!\left[\frac{P \triangleright ?\Gamma, \Gamma_1 \qquad Q \triangleright ?\Gamma, \Gamma_2}{P \parallel Q \triangleright ?\Gamma, \Gamma_1, \Gamma_2}\right]\!\right] = c_{[\![T]\!]} \odot ([\![P]\!] \parallel [\![Q]\!])$$

$$\left[\!\left[\frac{k \in I \quad P \triangleright \Gamma, \vec{x} :: T_k}{a \oplus \ell_k[\vec{x}].\, P \triangleright \Gamma, a : \oplus_{i\in I} \ell_i \cdot T_i}\right]\!\right] = (\iota_k : [\![T_k]\!] \multimap [\![\oplus_{i\in I} \ell_i \cdot T_i]\!]) \odot (\varphi^*_{\vec{x}::T_k}([\![P]\!]) : [\![\Gamma]\!]^\perp \multimap [\![T_k]\!])$$

$$\left[\!\left[\frac{\forall i : I, P_i \triangleright \Gamma, \vec{x} :: T_i}{a \,\&\, \{\ell_i(\vec{x}_i).\, P_i\}_{i\in I} \triangleright \Gamma, a : \&_{i\in I} \ell_i \cdot T_i}\right]\!\right] = \langle [\![\varphi^*_{\vec{x}_i::T_i}(P_i)]\!] : [\![\Gamma]\!]^\perp \multimap [\![T_i]\!] \rangle_{i\in I}$$

$$\left[\!\left[\frac{P \triangleright a : T, b : T^\perp, \Gamma}{(\nu\, a\, b : T) P \triangleright \Gamma}\right]\!\right] = ([\![P]\!] : ([\![T]\!]^\perp \parallel [\![T]\!]) \multimap [\![\Gamma]\!]) \odot (\alpha_{[\![T]\!]} : \emptyset \multimap [\![T]\!] \parallel [\![T]\!]^\perp)$$

$$\left[\!\left[\frac{P \triangleright ?\Gamma, \vec{x} :: T}{!a(\vec{x}).\, P \triangleright ?\Gamma, a : !T}\right]\!\right] = (!\varphi^*_{\vec{x}::T}([\![P]\!]) : !![\![\Gamma]\!]^\perp \multimap ![\![T]\!]) \odot \mu_{[\![\Gamma]\!]^\perp}$$

$$\left[\!\left[\frac{P \triangleright \Gamma, \vec{x} :: T}{?a[\vec{x}].\, P \triangleright \Gamma, a : ?T}\right]\!\right] = d_{[\![T]\!]} \odot (\varphi^*_{\vec{x}::T}([\![P]\!]) : [\![\Gamma]\!]^\perp \multimap [\![T]\!])$$

$$\left[\!\left[\frac{P \triangleright \Gamma, a : !T, \vec{x} :: T}{\#a(\vec{x}).\, P \triangleright \Gamma, a : !T}\right]\!\right] = \bar{d}_{[\![T]\!]} \odot \left(\varphi^*_{\vec{x}::T}([\![P]\!]) : [\![\Gamma]\!]^\perp \multimap [\![!T]\!] \parallel [\![T]\!]\right)$$

Fig. 11. Interpretation of π_{DiLL} into **Str**

Then, if i has won the race, it means that any other $j \neq i$ lost. In that case, messages on the j component are forwarded to the leftmost $!A$: $\mathsf{Req}^-_j \multimap \mathsf{Req}^+_{j,i}$ and $*_i \multimap \mathsf{Req}^+_{j,i}$. From there, we have a copycat strategy between the ith copy of A on the left and on the right. Formally, the event structure C_A is defined as follows. **Events** are of one of the following form:

- an event Req^-_i for $i \in \mathbb{N}$ mapped to the initial move of the right $!A$;
- an internal event $*_i$, representing the fact that the ith request has won the race
- an event $\mathsf{Req}^+_{i,j}$ for $i \neq j$, corresponding to the forwarding of Req^-_i when j wins the race;
- an event $\langle e, i\rangle$ for every $e \in \mathbb{C}_A$ and $i \in \mathbb{N}$ representing the forwarding to A when the race is won by i;
- an event $\langle e, i, j\rangle$ for every distinct numbers $i, j \neq \mathbb{N}$ and $e \in \mathbb{C}_A$, for forwarding the ith copy to $!A$ when i loses the race to j.

Causality includes the usual causal order on the copies of $\mathbb{C}_A$, plus the following links:

- $\mathsf{Req}^-_i \multimap *_i$ and $*_i \leq \langle e, i\rangle$ for all $e \in \mathbb{C}_A$;
- $\mathsf{Req}^-_i \multimap \mathsf{Req}^+_{i,j}$ and $*_j \multimap \mathsf{Req}^+_{i,j}$ for any $i \neq j$: $\mathsf{Req}^+_{i,j}$ is played when Opponent made the i-th request and j won the race; and
- $\mathsf{Req}^+_{i,j} \leq \langle e, i, j\rangle$ for $e \in \mathbb{C}_A$.

Conflict is generated by asking that the $*_i$ are all in mutual conflict.

C_A might not be receptive if A is positive (as in the example), so that the strategy $\bar{d}_A$ is obtained by precomposing with copycat to ensure receptivity. The interpretation of finite processes is given in Figure 11. Infinite processes are obtained as limits of chain of interpretation of their approximants, since the interpretation is easily seen to be monotonic.

We start by showing the adequacy of our translation. To state it, we need a notion of transition on strategies. Event structures come with a natural notion of transition system, given by configurations. Given an event structure E and $x \in \mathcal{C}(E)$, we define E/x as featuring those events $e \in E \setminus x$ that are not in conflict with any events in x, with causality and conflict directly inherited from E. This can be lifted to the level of strategies. Given a uniform strategy $\sigma : S \rightharpoonup \mathcal{A}$, and a configuration $x \in \mathcal{C}(S)$, we build the uniform strategy $\sigma/x : \mathcal{S}/x \rightharpoonup \mathcal{A}/\sigma\, x$ (read σ after x). We say that $\sigma : A$ can do a transition to τ with visible actions $y \in \mathcal{C}(A)$, written $\sigma \xrightarrow{y} \tau$, when there exists

a configuration $x \in C(\sigma)$ such that $\sigma/x \cong \tau$ and $\sigma\, x \cong y$ in $\tilde{A}$. The particular case where $y = \emptyset$ corresponds to an internal transition and is simply written $\sigma \longrightarrow \tau$.

Lemma 6.1. *Consider $P \triangleright \Delta$.*
(1) (a) *If $P \equiv Q$, then $[\![P]\!] \cong [\![Q]\!]$; and* (b) *If $P \longrightarrow Q$ then $[\![P]\!] \longrightarrow [\![Q]\!]$*
(2) If $P \equiv a \oplus \ell[\vec{x}].\, Q$, then there exists $s \in \min([\![P]\!])$ mapped to (a, ℓ) and $[\![P]\!]/\{s\} \cong [\![Q]\!]$.
(3) If $P \equiv ?a[\vec{x}].\, Q$, then there exists $s \in \min([\![P]\!])$ mapped to (a, Req) and $[\![P]\!]/\{s\} \cong [\![Q]\!]$.

Lemma 6.2 (Adequacy). *Consider a process $P \triangleright \Delta$. We have:*
(1) If $[\![P]\!] \longrightarrow \sigma$, then there exists $Q \triangleright \Delta$ with $P \longrightarrow^* Q$ and $[\![Q]\!] \cong \sigma$.
(2) If $s \in [\![P]\!]$ is a minimal positive event mapped to $(a, v) \in [\![\Delta]\!]$ then (a) *if $v = \mathsf{Req}$, then $P \equiv ?a[\vec{x}].\, Q$ and $[\![Q]\!] \cong \sigma/\{s\}$; and* (b) *if $v = \ell$, then $P \equiv a \oplus \ell[\vec{x}].\, Q$ and $[\![Q]\!] \cong \sigma/\{s\}$.*

6.2 Full Abstraction

Using the notion of transition on strategies defined in the previous section, we define weak bisimulations between strategies:

Definition 6.3. A *weak bisimulation* is an equivalence relation $\mathcal{R}$ between uniform strategies on the same tcg such that if $\sigma \mathcal{R} \tau$, for all configuration $y \in C(A)$, if $\sigma \xrightarrow{y} \sigma'$, then there exists τ' such that $\tau \xrightarrow{y} \tau'$ with $\sigma' \mathcal{R} \tau'$. We write $\approx$ for the largest weak bisimulation.

Lemma 6.4. *Weak bisimilarity is a congruence (stable under composition), hence $\sigma \cong \tau$ implies $\sigma \approx \tau$.*

From this result, and using the standard method of action testers [Hennessy 2007], we derive:

Theorem 6.5 (Second-order Full Abstraction). *Consider $P, Q \triangleright \Delta$. Then we have $P \cong_{\pi_{DiLL}} Q$ iff $[\![P]\!] \approx [\![Q]\!]$.*

6.3 Causal Semantics of ML$_{\|}$

By composing the syntactic translation and the semantics interpretation, we get an interpretation of types and programs of ML$_{\|}$ in terms of event structures: $[\![\sigma]\!] = [\![(\!|\sigma|\!)]\!]$ and $[\![M]\!] = [\![(\!|M|\!)]\!]$. From the previous results, we immediately get:

Lemma 6.6 (Adequacy). *Consider two terms $\Delta \vdash M, N : \sigma$. If $[\![M]\!] \approx [\![N]\!]$, then $M \cong_{ML_{\|}} N$.*

As we discussed in § 4.4, the converse does not hold (see Appendix B.4). However, we have *second-order full abstraction*. An interface $\Delta \vdash \sigma$ is second-order when (1) σ is a base type and (2) types in Δ are either base types, or arrow types between base types. We have:

Theorem 6.7 (Full Abstraction). *Consider two terms M, N well-typed on a second-order interface $\Delta \vdash \sigma$. Then $M \cong_{ML_{\|}} N$ if and only if $[\![M]\!] \approx [\![N]\!]$.*

This theorem is useful when applied to first-order functions calling other external first-order functions (from say, libraries).

7 IMPLEMENTATION

To illustate the causal model, we have used the model presented in this paper to implement a causal interpretation of a subset of OCaml corresponding to ML$_{\|}$ into event structures, representing strategies. The metalanguage process is built implicitly but not explicitly represented as they are not very informative and extremely verbose. The (anonymised) prototype is a web application, available at:

http://programminggamesemantics.github.io/index.html

The prototype allows entering OCaml code restricted to functions, product, record types and sum types. The standard library of OCaml is replaced by a simple kernel implemented by specific strategies to represent integer references (called `var`) and parallelism. See the webpage for details. The event structure is not displayed in its entirety as it would often be infinite (because of infinite datatypes and recursion) but instead the user can explore interactively branches of the computation they are interested in by clicking on Program events to unlock the next step of computation.

8 RELATED WORK

Metalanguage and Process Representation for Strategies and Games. Hyland and Ong [1995] first studied a relationship between game semantics and the π-calculus, where π-calculus processes are used to denote plays of innocent strategies (for PCF). This idea led to recast the traditional encoding of the call-by-value λ-calculus into the π-calculus [Milner 1992b] into a game semantics model for call-by-value PCF [Honda and Yoshida 1999]. In the sequential setting, the work by Longley [2009] proposed a programming language to describe sequential innocent strategies as a whole. Later, Goyet [2013] proposed an abstract calculus for sequential strategies, close to the πI-calculus.

Dimovski and Lazic [2004]; Ghica and Murawski [2006] represent strategies as CSP terms for use with model checkers, but CSP is only used a way to represent strategies, rather than a target language for syntactic translations - in particular all reasoning must be done at the level of the model. In a similar vein, Disney and Flanagan [2015] argue for a reading of strategies in terms of processes in the sequential setting, for type soundness.

Game Semantics for Concurrency and Nondeterminism. The first concurrent game semantics model, based on traces, is due to Laird [2001] for a message-passing language, extended to call-by-name shared-memory later by Ghica and Murawski [2004], angelic models which are fully abstract for may-equivalence. Causal models arrived later with [Sakayori and Tsukada 2017] (asynchronous π-calculus) and [Castellan and Clairambault 2016] (IPA: call-by-name shared-memory), which are both angelic (ie. only adequate for may-testing). Harmer and McCusker [1999] provide the first non-angelic model of game semantics in the sequential setting based on *stopping traces*. This approach is tailored to must-equivalence.

Our approach thus gives the first non-angelic game model of a realistic higher-order concurrent programming language, capturing faithfully the nondeterministic branching behaviours of shared-memory concurrent programs.

Recently, Melliès [2019] gave a games semantics model for DiLL based on templates games. It differs from ours in two ways: (1) his model is synchronous, while our model is asynchronous (due to courtesy); and (2) it ignores deadlocks, preventing from modelling ML$_\parallel$ adequately. Melliès and Stefanesco [2020] extended it later to model Concurrent Separation Logic.

Extensions of the Linear-Logic and Session Types Correspondence. In this paper, we use a variation of Differential Linear Logic (DiLL) to express nondeterminism built on deterministic computations. Beffara [2006] presents a model of Linear Logic in terms of processes of the πI-calculus, used to interpret concurrent extensions of the λ-calculus. In the context of proof-nets, Ehrhard and Laurent [2010] investigate encoding a finitary π-calculus in a transition system of labelled differential interaction nets.

Following the work by Toninho et al. [2012] on an encoding of the simply-typed λ-calculus, Toninho and Yoshida [2018] have proven that there exist *mutually inverse* and *fully abstract* encodings between a polymorphic session π-calculus in [Caires et al. 2013] and a linear formulation of System F. To gain expressiveness beyond functional and strong normalising behaviours, several extensions have been proposed, eg. the lock primitives for nondeterminism [Balzer and Pfenning 2017]; dynamic monitoring [Gommerstadt et al. 2018; Jia et al. 2016]; exceptional handling [Caires and Pérez 2017]; multiparty interactions [Carbone et al. 2016, 2015]; hyperenvironments to capture non-blocking I/O [Kokke et al. 2019a]; and a bounded linear logic based-extension to model racy conditions [Kokke et al. 2019b]. Balzer et al. [2018] show an encoding of untyped asynchronous communication in [Balzer and Pfenning 2017], and Balzer et al. [2019] successfully enabled deadlock-freedom by introducing extra-logically imposed partial orders.

Our approach differs from them, aiming to *describe* game semantics translations more accurately by the metalanguage arisen from event structures grounded on DiLL. None of the above calculi provides a fully abstract encoding from nondeterministic and concurrent languages (ie. ML$_\parallel$) nor a causal semantic interpretation.

9 CONCLUSION

Our contribution in this paper is twofold: we have presented a factorisation of the usual game semantics methodology in two steps: (1) a syntactic translation and (2) a semantics interpretation which allows for separating concerns when designing new game semantics models. We have applied this methodology to build the first non-angelic interactive model of a call-by-value language with shared memory concurrency. We have used our framework to implement a causal modelling of a subset of a concurrent extension of OCaml. We believe our methodology proves promising to model mainstream programming languages and systems where semanticists will need inputs from language or system designers who do not have an immediate access to game semantics.

Another avenue would be to use the model to reason about open terms, and inserts itself naturally in some new research programs such as [Gu et al. 2018; Koenig and Shao 2020] which aim at building large certified systems by assembling components proved correct in isolation.

Other future work include relaxed shared memory using the techniques presented in [Castellan 2016], and trying to characterise **UStr** (or a subset of it) as an initial category.

REFERENCES

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full abstraction for PCF. *Information and Computation* 163, 2 (2000), 409–470.

Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *PACMPL* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 611–639.

Emmanuel Beffara. 2006. A Concurrent Model for Linear Logic. *Electr. Notes Theor. Comput. Sci.* 155 (2006), 147–168. https://doi.org/10.1016/j.entcs.2005.11.055

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *ESOP*. 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science)*, Paul Gastin and François Laroussinie (Eds.), Vol. 6269. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schuermann, and Philip Wadler. 2016. Coherence Generalises Duality: a logical explanation of multiparty session types. In *CONCUR'16 (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 59. Schloss Dagstuhl, 33:1–33:15.

Marco Carbone, Fabrizio Montesi, Carsten Schormann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl, 412–426.

Giuseppe Castagna and Luca Padovani. 2009. Contracts for Mobile Processes. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings (Lecture Notes in Computer Science)*, Mario Bravetti and Gianluigi Zavattaro (Eds.), Vol. 5710. Springer, 211–228. https://doi.org/10.1007/978-3-642-04081-8_15

Simon Castellan. 2016. Weak memory models using event structures. In *Vingt-septième Journées Francophones des Langages Applicatifs (JFLA 2016)*.

Simon Castellan. 2017. *Concurrent structures in game semantics*. Ph.D. Dissertation. ENS Lyon, France.

Simon Castellan and Pierre Clairambault. 2016. Causality vs interleaving in game semantics. In *CONCUR 2016 - Concurrency Theory*.

Simon Castellan, Pierre Clairambault, Jonathan Hayman, and Glynn Winskel. 2018. Non-angelic Concurrent Game Semantics. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 3–19.

Simon Castellan, Pierre Clairambault, and Glynn Winskel. 2014. Symmetry in Concurrent Games. In *CSL-LICS 2014*. IEEE Computer Society.

Simon Castellan, Pierre Clairambault, and Glynn Winskel. 2019. Thin Games with Symmetry and Concurrent Hyland-Ong Games. *Logical Methods in Computer Science* 15, 1 (2019). https://lmcs.episciences.org/5248

Simon Castellan and Nobuko Yoshida. 2019. Two Sides of the Same Coin: Session Types and Game Semantics. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2019, Lisbon, Portugal*.

Aleksandar Dimovski and Ranko Lazic. 2004. CSP Representation of Game Semantics for Second-Order Idealized Algol. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings (Lecture Notes in Computer Science)*, Jim Davies, Wolfram Schulte, and Michael Barnett (Eds.), Vol. 3308. Springer, 146–161. https://doi.org/10.1007/978-3-540-30482-1_18

Tim Disney and Cormac Flanagan. 2015. Game Semantics for Type Soundness. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 104–114. https://doi.org/10.1109/LICS.2015.20

Thomas Ehrhard. 2018. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science* 28, 7 (2018), 995–1060. https://doi.org/10.1017/S0960129516000372

Thomas Ehrhard and Olivier Laurent. 2010. Interpreting a Finitary Pi-Calculus in Differential Interaction Nets. *Information and Computation* 208, 6 (June 2010), 606–633.

Dan R. Ghica and Andrzej S. Murawski. 2004. Angelic Semantics of Fine-Grained Concurrency. In *Foundations of Software Science and Computation Structures*, Igor Walukiewicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–225.

Dan R. Ghica and Andrzej S. Murawski. 2006. Compositional Model Extraction for Higher-Order Concurrent Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science)*, Holger Hermanns and Jens Palsberg (Eds.), Vol. 3920. Springer, 303–317. https://doi.org/10.1007/11691372_20

Dan R. Ghica and Nikos Tzevelekos. 2012. A System-Level Game Semantics. *Electr. Notes Theor. Comput. Sci.* 286 (2012), 191–211. https://doi.org/10.1016/j.entcs.2012.08.013

Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *ESOP*. 771–798. https://doi.org/10.1007/978-3-319-89884-1_27

Alexis Goyet. 2013. The Lambda Lambda-Bar calculus: a dual calculus for unconstrained strategies. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 155–166. https://doi.org/10.1145/2429069.2429089

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Russell Harmer and Guy McCusker. 1999. A Fully Abstract Game Semantics for Finite Nondeterminism. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 422–430. https://doi.org/10.1109/LICS.1999.782637

Matthew Hennessy. 2007. *A Distributed Pi-Calculus*. CUP.

Kohei Honda and Nobuko Yoshida. 1995. On Reduction-Based Process Semantics. *TCS* 151, 2 (1995), 437–486.

Kohei Honda and Nobuko Yoshida. 1999. Game-Theoretic Analysis of Call-by-Value Computation. *TCS* 221 (1999), 393–456.

J. M. E. Hyland and C.-H. Luke Ong. 1995. Pi-Calculus, Dialogue Games and PCF. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. 96–107. https://doi.org/10.1145/224164.224189

Martin Hyland and Luke Ong. 2000. On full abstraction for PCF. *Information and Computation* 163 (2000), 285–408.

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and Blame Assignment for Higher-order Session Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 582–594. https://doi.org/10.1145/2837614.2837662

Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proc. 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'20)*.

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019a. Better Late Than Never: A Fully-abstract Semantics for Classical Processes. *Proc. ACM Program. Lang.* 3, POPL, Article 24 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290337

Wen Kokke, J. Garrett Morris, and Philip Wadler. 2019b. Towards Races in Linear Logic. In *Coordination (Lecture Notes in Computer Science)*, Vol. 11533. Springer, 37–53. https://doi.org/10.1007/978-3-030-22397-7_3

James Laird. 2001. A Game Semantics of Idealized CSP. *Electr. Notes Theor. Comput. Sci.* 45 (2001), 232–257. https://doi.org/10.1016/S1571-0661(04)80965-4

John Longley. 2009. Some Programming Languages Suggested by Game Models (Extended Abstract). *Electr. Notes Theor. Comput. Sci.* 249 (2009), 117–134. https://doi.org/10.1016/j.entcs.2009.07.087

Paul-André Melliès and Léo Stefanesco. 2020. Concurrent separation logic meets template games. In *Proc. 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'20)*.

P. Melliès. 2019. Template games and differential linear logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13.

Paul-André Melliès. 2003. Asynchronous games 1: A group-theoretic formulation of uniformity. *Manuscript, Available online* (2003).

Paul-André Melliès. 2009. Categorical semantics of linear logic. In *Interactive models of computation and program behaviour*. Société mathématique de France.

Robin Milner. 1992a. Functions as Processes. *Math. Struct. Comput. Sci.* 2, 2 (1992), 119–141. https://doi.org/10.1017/S0960129500001407

Robin Milner. 1992b. Functions as Processes. *MSCS* 2, 2 (1992), 119–141.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Robin Milner and Davide Sangiorgi. 1992. Barbed Bisimulation *(Lecture Notes in Computer Science)*, W. Kuich (Ed.), Vol. 623. 685–695.

Silvain Rideau and Glynn Winskel. 2011. Concurrent Strategies. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. 409–418. https://doi.org/10.1109/LICS.2011.13

Ken Sakayori and Takeshi Tsukada. 2017. A Truly Concurrent Game Model of the Asynchronous π-Calculus. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*. Springer-Verlag New York, Inc., New York, NY, USA, 389–406. https://doi.org/10.1007/978-3-662-54458-7_23

Davide Sangiorgi. 1996. pi-Calculus, Internal Mobility, and Agent-Passing Calculi. *Theor. Comput. Sci.* 167, 1&2 (1996), 235–274. https://doi.org/10.1016/0304-3975(96)00075-8

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 275–287. https://doi.org/10.1145/2676726.2676985

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2012. Functions as Session-Typed Processes. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 346–360. https://doi.org/10.1007/978-3-642-28729-9_23

Bernardo Toninho and Nobuko Yoshida. 2018. On Polymorphic Sessions And Functions: A Tale of Two (Fully Abstract) Encodings. In *27th European Symposium on Programming (LNCS)*, Vol. 10801. Springer, 827–855.

Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418. https://doi.org/10.1017/S095679681400001X

Glynn Winskel. 1986. Event Structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986.* 325–392. https://doi.org/10.1007/3-540-17906-2_31

Glynn Winskel. 2007. Event structures with symmetry. *Electronic Notes in Theoretical Computer Science* 172 (2007), 611–652.

A APPENDIX OF SECTION 3

A.1 Proofs

Lemma 3.2 (Subject Reduction). *If $P \triangleright \Gamma$ and $P \longrightarrow Q$, then $Q \triangleright \Gamma$.*

Proof. **Case (1: [race]))** Assume $P_0 = (?a[\vec{x}].\, P \parallel \#b(\vec{y}).\, Q \parallel R)$ and

$$P_0 \triangleright \Gamma, a :?A, b : A^{\perp} \tag{1}$$

Then by the intervion of [par], we have:

$$?a[\vec{x}].\, P \triangleright \Gamma_1, ?\Gamma_0, a :?A \quad \text{and} \quad \#b(\vec{y}).\, Q \triangleright \Gamma_2, ?\Gamma_0, b : A^{\perp} \quad \text{and} \quad R \triangleright \Gamma_3, ?\Gamma_0 \quad \text{with} \quad \Gamma = \Gamma_1, \Gamma_2, \Gamma_3, ?\Gamma_0 \tag{2}$$

From (2), and inversion of [req] and [nd], we have

$$P \triangleright \Gamma_1, ?\Gamma_0, a :?A, \vec{x} :: T \quad \text{and} \quad Q \triangleright \Gamma_2, ?\Gamma_0, b : A^{\perp}, \vec{y} :: T' \text{ with } A^{\perp} = !T',\ T' = T^{\perp} \tag{3}$$

From (3), applying [par] and [res], we have

$$(\nu \vec{x}\vec{y})(P \parallel Q) \triangleright \Gamma_1, a : A, \Gamma_2, ?\Gamma_0, b : A^{\perp} \tag{4}$$

Applying [par] to (4) and (2), we have:

$$R \parallel (\nu \vec{x}\vec{y})(P \parallel Q) \triangleright \Gamma_1, a : A, \Gamma_2, \Gamma_3, ?\Gamma_0, b : A^{\perp} \tag{5}$$

Applying [res] to (5), we obtain the result.

Case (2: [cxt] and [str])) Straightforward by IH. □

B PROOFS OF § 4

We prove here Theorem 4.7.

B.1 Approximation and normal form

Consider a finite semiclosed term M. Because there are no fixpoint, we know that M must reduce to a normal form $\mathsf{nf}(M)$ for $\longrightarrow$ (since we are simply in the simply-typed λ-calculus with uninterpreted constants). By Lemma 4.4, we know that $(\!|M|\!)_{c,o} \equiv (\!|\mathsf{nf}(M)|\!)_{c,o}$. A term is **stuck** when it is in normal form but not a value. Any finite term M has a normal form written $\mathsf{nf}(M)$. We say that a machine is finite when the term it contains is finite. In that case, we extend the notation $\mathsf{nf}(\cdot)$: $\mathsf{nf}(\Delta \vdash^{\vec{y}} \langle M, \mu\rangle) := \Delta \vdash^{\vec{y}} \langle \mathsf{nf}(M), \mu\rangle$. If σ is a base type, we write $\mathsf{nf}(\sigma)$ for the set of normal forms of semiclosed terms of type σ.

Lemma B.1. *The set of semiclosed stuck terms is contained in the following grammar:*

$$S ::= (\lambda \vec{x}.\, M)\, S\, N\, \ldots\, N \mid \mathtt{if}\, S\, M\, M \mid \mathtt{ref}(\underline{n}) \mid\, !r \mid r := \underline{n}.$$

where M means any term, and N means normal form (not necessarily stuck).

Proof. Consider a stuck term S. We proceed by induction on its syntax.

- It cannot be a λ-abstraction since semiclosed terms do not have arrow types.

- If it is a conditional $\texttt{if}\, M\, N\, N'$, then M must be normal as otherwise S would not be normal. If M was a value, then it would have to be an immediate boolean since semiclosed terms do not have free variables of type bool. But then, S would reduce to N or N', which is absurd. So M must be stuck.
- If it is a variable: impossible, variable are values.
- If it is application: then S has the form $M\, N_1\, \ldots\, N_k$ where M is not an application. Since S is semiclosed, M can only be an abstraction or a constant.
  - If it is an abstraction: then all the N_i must be in normal form. Moreover, the first argument must not be a value otherwise, there would be a reduction.
  - If it a constant, then we see that the only possible cases are $\texttt{ref}(\underline{n})$, $!r$, or $r := \underline{n}$.

□

B.2 Finite adequacy

Lemma B.2. *For any semiclosed value V, $\Delta \vdash^{\vec{y}} \langle V, \mu\rangle$ cannot do any transition.*

Proof. Easy inspection. □

Lemma B.3 (Finite adequacy). *Consider a finite machine $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle$. If $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!) \xrightarrow{\alpha} P$, then $\alpha = \tau$, and there are two cases:*

- *Either $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\tau} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ with $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!) \equiv P$,*
- *Or $P \equiv \alpha.\, Q$ and $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ with $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!) \equiv Q$.*

Proof. First, by Lemma 4.4, we can assume without loss of generality that M is in normal form. Moreover, given the shape of $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)$ it is clear that α must be τ as there are no visible actions at top level. By Lemma B.2, we know that M must be stuck. We examine the cases given by Lemma B.1:

- If $M = \texttt{ref}(\underline{n})$, then we have a contradiction as this term cannot reduce.
- The cases for $!r$, are $r := \underline{n}$ are straightforward calculation. For instance, assume that $M = !r$. In this case, there are only two possible reductions: indeed $(\!|M|\!)_{c,o}$ starts with a dereliction that has to communicate with the codereliction on r. That must have caused the τ-transition. Then there are two cases, whether $r \in \vec{y}$ or not.
  - If $r \notin \vec{y}$: then the read is performed, and the value is sent back to M, and we have $P \equiv (\!|\Delta \vdash^{\vec{y}} \langle \underline{\mu(n)}, \mu\rangle|\!)$ as desired.
  - Otherwise, r is visible and there is a visible transition on I in front of the continuation, and we end up on the same process after performing it.
- If $M = \texttt{if}\, M_0\, N\, N'$, then we know that M_0 must be stuck. By construction of the translation, we must have $\Delta \vdash^{\vec{y}} \langle M_0, \mu\rangle \xrightarrow{\alpha} P'$ and P can be obtained from P' by the interpretation of if. Then we can apply the induction hypothesis.
- If $M = (\lambda \vec{x}.\, M_0)\, \vec{N}$: then once again by the definition of the interpretation the reduction must occur within one of the N_i, and we conclude by induction.

□

B.3 Conclusion

We can now conclude the adequacy result and the correctness result of our translation of machines.

Lemma 4.6 (Adequacy). *If $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \xrightarrow{\alpha} P$, then $\alpha = \tau$, and there are two cases:*

- *Either $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\tau} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ with $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!)_{\mathrm{I}} \equiv P$,*

- *Or* $P \equiv \alpha^{\mathrm{I}}.\, Q$ *and* $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\alpha} \Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle$ *with* $(\!|\Delta' \vdash^{\vec{y}'} \langle N, \mu'\rangle|\!)_{\mathrm{I}} \equiv Q$.

Proof. Write $\mathfrak{m}$ for the input machine. Clearly $\alpha = \tau$ for the same reason as in Lemma B.3. Consider now a finite approximation P_0 of P: by definition of the LTS, there exists a finite approximation Q_0 of $(\!|m|\!)$ with $Q_0 \xrightarrow{\tau} P_0$. By continuity, we can find a finite approximation M_n of M such that $Q_0 \leq (\!|\Delta \vdash^{\vec{y}} \langle M_n, \mu\rangle|\!) \leq (\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle_n|\!)$. Then, we can apply Lemma B.3, and do a case distinction — the two cases are similar. Say that $\Delta \vdash^{\vec{y}} \langle M_n, \mu\rangle \xrightarrow{\tau} \Delta' \vdash^{\vec{y}'} \langle R_n, \mu'\rangle$ with $P_0 \equiv\leq (\!|\Delta' \vdash^{\vec{y}'} \langle R_n, \mu'\rangle|\!) \leq\equiv P$. This implies that $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \xrightarrow{\tau} \Delta' \vdash^{\vec{y}'} \langle R, \mu'\rangle$. Then it is easy to see that $(\!|\Delta' \vdash^{\vec{y}'} \langle R, \mu'\rangle|\!)$ is the limit of the $(\!|\Delta' \vdash^{\vec{y}'} \langle R, \mu'\rangle_n|\!)$, hence $(\!|\Delta \vdash^{\vec{y}} \langle R, \mu\rangle|\!) \equiv P$ as desired. □

Theorem 4.7 (Soundness and completeness). *For any typed* $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle$ *and* $\Delta' \vdash^{\vec{y}} \langle N, \mu'\rangle$*, the following are equivalent:* (1) $\Delta \vdash^{\vec{y}} \langle M, \mu\rangle \approx_{\mathsf{ML}_{\|}} \Delta \vdash^{\vec{y}} \langle N, \mu'\rangle$, (2) $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \approx_{\pi_{DiLL}} (\!|\Delta \vdash^{\vec{y}} \langle N, \mu'\rangle|\!)_{\mathrm{I}}$ *and* (3) $(\!|\Delta \vdash^{\vec{y}} \langle M, \mu\rangle|\!)_{\mathrm{I}} \cong_{\pi_{DiLL}} (\!|\Delta \vdash^{\vec{y}} \langle N, \mu'\rangle|\!)_{\mathrm{I}}$

Proof. (1) ⇔ (2): To show the result, it is enough to show that

$$\mathcal{R}_0 = \{((\!|\mathfrak{m}|\!), (\!|\mathfrak{n}|\!)), (\alpha^{\mathrm{I}}.\, (\!|\mathfrak{m}|\!), \alpha^{\mathrm{I}}.\, (\!|\mathfrak{n}|\!)) \mid \mathfrak{m} \approx_{\mathsf{ML}_{\|}} \mathfrak{n}\}$$
$$\mathcal{R}_1 = \{(\mathfrak{m}, \mathfrak{n}) \mid (\!|\mathfrak{m}|\!) \approx_{\pi_{\mathrm{DiLL}}} (\!|\mathfrak{n}|\!)\}$$

are weak bisimulations.

(1) For $\mathcal{R}_0$:
- If $(\alpha^{\mathrm{I}}.\, (\!|\mathfrak{m}|\!), \alpha^{\mathrm{I}}.\, (\!|\mathfrak{n}|\!)) \in \mathcal{R}_0$, then it is easy to see that they both can only do α^{I} before getting to $((\!|\mathfrak{m}|\!), (\!|\mathfrak{n}|\!))$ which is still in $\mathcal{R}_0$.
- If $((\!|\mathfrak{m}|\!), (\!|\mathfrak{n}|\!)) \in \mathcal{R}_0$: if $(\!|\mathfrak{m}|\!) \xrightarrow{\alpha} P$, then we know that $\alpha = \tau$ by Lemma 4.6, and there are two cases:
  - If $\mathfrak{m} \xrightarrow{\tau} \mathfrak{m}'$ with $(\!|\mathfrak{m}'|\!) \equiv P$, then $\mathfrak{n} \xrightarrow{\tau} \mathfrak{n}'$ for some $\mathfrak{n}'$ with $\mathfrak{m}' \approx_{\mathsf{ML}_{\|}} \mathfrak{n}'$ and we conclude via Lemma 4.5.
  - If $\mathfrak{m} \xrightarrow{\alpha} \mathfrak{m}'$ and $(\!|\mathfrak{m}'|\!) \equiv \alpha^{\mathrm{I}}.\, P$, then $\mathfrak{n} \xrightarrow{\alpha} \mathfrak{n}'$ with $\mathfrak{m}' \approx_{\mathsf{ML}_{\|}} \mathfrak{n}'$. By Lemma 4.5, we know that $(\!|\mathfrak{n}|\!) \xrightarrow{\alpha} Q$ and $Q \equiv \alpha^{\mathrm{I}}.\, (\!|\mathfrak{n}'|\!)$ hence $(P, Q) \in \mathcal{R}_0$ as desired.

(2) For $\mathcal{R}_1$: Assume that $(\mathfrak{m}, \mathfrak{n}) \in \mathcal{R}_1$, and $\mathfrak{m} \xrightarrow{\alpha} \mathfrak{m}'$. Then by Lemma 4.5, we have $(\!|\mathfrak{m}|\!) \xRightarrow{\alpha} (\!|\mathfrak{m}'|\!)$. By construction, this implies that $(\!|\mathfrak{n}|\!) \xRightarrow{\alpha} (\!|\mathfrak{n}'|\!)$ for some $\mathfrak{n}'$, and we conclude by Lemma 4.6.

(2) ⇔ (3) For this, we need the full abstraction result of Theorem 6.5. Indeed, it is easy to see that machines $\mathfrak{m}, \mathfrak{m}'$, we have

$$[\![\mathfrak{m}]\!] \approx_{\mathsf{ML}_{\|}} [\![\mathfrak{m}']\!] \Leftrightarrow [\![\mathfrak{m}]\!] \approx [\![\mathfrak{m}']\!].$$

□

B.4 Discussion on full abstraction

In this subsection, we offer a detailed discussion on the full abstraction. This is a well-known phenomenon in game semantics due to some information that is lost during the translation: the difference between calling and returning. At the level of the metalanguage, the distinction is gone which allows us to write contexts that can for instance, evaluate an argument *and* return in parallel which is not possible in $\mathsf{ML}_{\|}$.

We now show a concrete examples, using the following two terms:

$$\begin{array}{rcl} M & = & f(\lambda x.()); \bot \\ N & = & \mathtt{let}\, r = \mathtt{ref}\, 0\, \mathtt{in}\, \mathtt{let}\, s = \mathtt{ref}\, 0\, \mathtt{in} \\ & & f(\lambda x.\, r := 1; s := 1); r := !r + 1; \\ & & \mathtt{if}(!r = 1\, \mathtt{and}\, !s = 0)\, \mathtt{then}\, ()\, \mathtt{else}\, \bot \end{array}$$

M and N are higher-order terms, with one higher order parameter $f : (\bullet \to \bullet) \to \bullet$. M calls its parameter on a dummy function, and then diverges. N does almost the same, except that there is a possibility for it converge: if at the end, r equals one and s zero. From the point of view of *Context*, the two arguments are equivalent, as they always return after being called. For N to converge, only r must be set to one, which is impossible to achieve using a context written in ML$_\parallel$ as calls to f will always finish completely before the control is passed back to N. However, there is a process that can interact differently with M and N. Indeed, their interpretations at o are processes over the context

$$\Gamma = o : \mathtt{Ret}()^+, f : ?(\mathtt{Call}(\lambda)^+ \cdot (!\mathtt{Call}()^- \cdot \mathtt{Ret}()^+ \parallel \mathtt{Ret}()^-))$$

On this interface, we can build a context that will trigger the events $\mathtt{Call}()^-$ and $\mathtt{Ret}()^-$ in parallel, which simulates a function that calls its argument **while** returning. In N, this triggers several races, including one between the write on s and the read on s: this race makes it possible for the condition to be evaluated after r is set to one, but before s is. Such a context can be written as follows:

$$C[] = (\nu o\bar{o})([] \mid \bar{o} \,\&\, \mathtt{Ret}().\, \alpha \mid !f(x).\, x \,\&\, \lambda(x, r).\, (?x[x_0].\, x_0 \oplus \mathtt{Call}() \parallel r \oplus \mathtt{Ret}())).$$

However as noted in the main part of the paper, we can derive the full abstraction result for the second-order (Theorem 6.7).

C PROOFS OF § 5

Lemma 5.15. *Any countable chain of embeddings $(f_i : \sigma_i \hookrightarrow \sigma_{i+1})_{i\in\mathbb{N}}$ has a colimit, ie. there exists a unique a strategy σ and embeddings $g_i : \sigma_i \hookrightarrow \sigma$ such that $g_{i+1} \circ f_i = g_i$.*

Proof. Write S_i for the underlying event structure of σ_i. We define

$$S = \bigcup_{i\in\mathbb{N}} S_i \setminus f_{i-1}(S_{i-1}),$$

with the convention that $f_0(f_0) = \emptyset$. Causality is defined as follows: $s \leq_S s'$ when there exists $i \in \mathbb{N}$ where both s, s' can be embeded and where $s \leq_{S_i} s'$. Similarly for conflict. It is easy to see that S_i embeds into S and that S is the smallest such event structure. Moreover, we can project S to A by simply using the σ_i. □

Lemma 5.8. *For every family of strategies $(\sigma_k : A \nrightarrow B_k)_{k\in K}$ there exists a strategy $\langle\sigma_k\rangle_{k\in K} : A \nrightarrow \&_{k\in K}\, \ell_k^- \cdot B_k$ such that $\pi_k \odot \langle\sigma_k\rangle_{k\in K} \cong \sigma_k$*

Proof. Without loss of generality, we can assume that each S_i contains the minimal strategy on $A^\perp \parallel B_i$. We define S to be the union of the S_i, plus events $\{\ell_i \mid i \in I\}$, with all courteous links from ℓ_i to events in S_i, and conflict that of the S_i plus all the minimal conflict between the ℓ_i. The mapping σ from S to $A^\perp \parallel \&_{k\in K}\ell_k^- \cdot B_k$ is given by the σ_i. It is easy to check that this strategy satisfies the desired axiom. □

C.1 Proof of the linear exponential comonad

Lifting and polarised categories. We define two operations on game: $\uparrow^+\mathcal{A} = \mathtt{Req}^+ \cdot \mathcal{A}$ and $\uparrow^-\mathcal{A} = \mathtt{Req}^- \cdot \mathcal{A}$. We define the polarised subcategories of **UStr**: **UStr**$^-$ is the category of negative games and negative strategies; while **UStr**$^+$ is the category of positive games and negative

strategies. Duality induces an isomorphism $(\mathbf{UStr}^-)^{\mathrm{op}} \cong \mathbf{UStr}^+$. This operation on tcgs extends to a functor $\uparrow^- - : \mathbf{UStr} \to \mathbf{UStr}^-$: given $\sigma : \mathcal{A} \nrightarrow \mathcal{B}$, $\uparrow^- \sigma$ starts by acknowledging the negative $\mathtt{Req}^-$ on the right, before playing $\mathtt{Req}^+$ on the left, and continuing as σ. As a result, $\uparrow^- \sigma$ is a negative strategy.

Lemma C.1. *The functor $\uparrow^-(-) : \mathbf{UStr} \to \mathbf{UStr}^-$ is a right adjunt to the inclusion $\mathbf{UStr}^- \subseteq \mathbf{UStr}$.*

Proof. The adjunction is fairly simple to describe:

- From $\sigma : \uparrow^+ A \nrightarrow B$ in $\mathbf{UStr}^+$, by removing the initial negative event on $\uparrow^+ A$, we get a strategy $A \nrightarrow B$ in $\mathbf{UStr}$ (not necessarily negative).
- From $\sigma : A \nrightarrow B$ in $\mathbf{UStr}$, we can build a strategy $\mathtt{Req}^- \cdot \sigma : \uparrow^+ A \nrightarrow B$ which is well-defined because B is positive.

It is clear that these operations are inverse of each other; naturality is a simple calculation. □

Lifting the monad. We are now ready to prove the result:

Lemma 5.14. *The construction $!(-)$ extends to an exponential comonad $\mathbf{UStr} \to \mathbf{UStr}$, that is it comes equipped a functorial action plus the following strategies:*

return: $\mathbf{d}_{\mathcal{A}} : !\mathcal{A} \nrightarrow \mathcal{A}$ ***digging:*** $\mu_{\mathcal{A}} : !\mathcal{A} \nrightarrow !!\mathcal{A}$ ***contraction:*** $\mathbf{c}_{\mathcal{A}} : !A \nrightarrow !A \parallel !A$.

satisfying standard laws [Melliès 2009].

Proof. From [Castellan 2017], we know that $\sharp \cdot$ is an exponential comonad on $\mathbf{UStr}^-$. We notice that $!A = \sharp \uparrow^- \mathcal{A}$ and we conclude by the fact that $\uparrow^- \mathcal{A}$ is a right adjunct to the inclusion: it thus transports exponential comonads on $\mathbf{UStr}^-$ to exponential comonads on $\mathbf{UStr}$ as desired. □

D PROOFS OF § 6

D.1 Adequacy

Lemma 6.1. *Consider $P \triangleright \Delta$.*

(1) (a) *If $P \equiv Q$, then $[\![P]\!] \cong [\![Q]\!]$; and* (b) *If $P \to Q$ then $[\![P]\!] \longrightarrow [\![Q]\!]$*
(2) If $P \equiv a \oplus \ell[\vec{x}].\, Q$, then there exists $s \in \min([\![P]\!])$ mapped to (a, ℓ) and $[\![P]\!]/\{s\} \cong [\![Q]\!]$.
(3) If $P \equiv ?a[\vec{x}].\, Q$, then there exists $s \in \min([\![P]\!])$ mapped to $(a, \mathtt{Req})$ and $[\![P]\!]/\{s\} \cong [\![Q]\!]$.

Proof. (1) To show the result on $\equiv$, since $\cong$ is a congruence, it is enough to show that it satisfies the rules listed in Figure 7.
The first block is a consequence of the monoidal structure and the compact-closure. The third block is a direct consequence of the weak product structure and the exponential comonad structure.
For the second block:

- [nil]: In the composition induced by (vab), the initial move is a negative move on a. That means that every positive move will be dependent on this negative move, in particular any positive move. This means that the composition can only contain negative events on channels distinct to a, b, so the process must be isomorphic to the empty strategy.
- [res]: Consequence of compact closure
- [id]: Consequence of the categorical setting: the forwarder behaves as an identity
- [swap]: If $c \neq \{a, b\}$, then we can safely push it outside the restriction: indeed, this will not create new causal links from the prefix on c to positive actions in P_i: such actions are already waiting on $\mathfrak{a}$. Hence, they are waiting on an action on b, that if happens, must occur in $\mathfrak{c}^-[Q_i]$, hence after $\mathfrak{c}^-$.

(2) If $P \equiv a \oplus \ell[\vec{x}].\, Q$, then we know that $[\![P]\!] \cong a \oplus \ell[\vec{x}].\, Q$. The interpretation of the latter is $\iota_k \odot [\![Q]\!]$: the strategy ι_k has (a, ℓ) as a minimal event which remains after composition since it is not in the hidden part. Similar reasoning for $P \equiv ?a[\vec{x}].\, Q$.

(3) If $P \longrightarrow Q$, then we proceed by induction on $\longrightarrow$. The rules [STR] follows from the previous point and rule [CTX] follows from the fact that evaluation contexts can not postpone a neutral event: any minimal neutral event of $[\![P]\!]$ is a minimal neutral event of $[\![E[P]]\!]$. The most interesting rule is [RACE]. This is a consequence of the construction of C_A: if $(\nu ab)(?a[\vec{x}].\, P \parallel a\#(\vec{y})Q \parallel R) \longrightarrow (\nu ab)(R \parallel (\nu \vec{x}\vec{y})(P \parallel Q))$, then the Req_i^+ emitted by the dereliction meets one initial Req_i^- of the codereliction. This synchronisation is internal and unlocks the desired neutral event $*_i$. Then, C_A behaves as copycat, which are the counterpart of the restrictions appearing in the right-hand-side of the syntactic rule.

□

Lemma 6.2 (Adequacy). *Consider a process $P \triangleright \Delta$. We have:*

(1) If $[\![P]\!] \longrightarrow \sigma$, then there exists $Q \triangleright \Delta$ with $P \longrightarrow^* Q$ and $[\![Q]\!] \cong \sigma$.

(2) If $s \in [\![P]\!]$ is a minimal positive event mapped to $(a, v) \in [\![\Delta]\!]$ then (a) *if $v = \mathsf{Req}$, then $P \equiv ?a[\vec{x}].\, Q$ and $[\![Q]\!] \cong \sigma/\{s\}$; and* (b) *if $v = \ell$, then $P \equiv a \oplus \ell[\vec{x}].\, Q$ and $[\![Q]\!] \cong \sigma/\{s\}$.*

Proof. (1) This reduction can be already be done by a finite approximant P_0 of P: $[\![P_0]\!] \longrightarrow \sigma_0$ and τ finite approximation of σ. We can proceed by induction on P_0.

- If $P_0 = P_1 \parallel P_2$: then the neutral event must come from $[\![P_1]\!]$ or $[\![P_2]\!]$, and we can conclude by induction.
- If P_0 starts with a negative prefix, then it does not have a neutral minimal event.
- If P_0 starts with a positive prefix, we conclude by induction.
- If P_0 is of the form $(\nu \vec{ab})P_1$: we consider the neutral event s of P_0. It corresponds to a neutral event s' of $[\![P_1]\!]$ which might not be minimal. Without loss of generality, we can assume that all deterministic reductions in P_1 are reduced. This means that P_1 must be of the form $a\#(\vec{x})Q \mid ?\bar{a}[\vec{y}].\, R$ where a and $\bar{a}$ are connected by a nu, and we can conclude.

(2) Same reasoning, by induction on a finite approximant.

□

D.2 Proof of full abstraction

Theorem 6.5 (Second-order Full Abstraction). *Consider $P, Q \triangleright \Delta$. Then we have $P \cong_{\pi_{DiLL}} Q$ iff $[\![P]\!] \approx [\![Q]\!]$.*

Proof. From Lemmata 6.4, 6.1, and 6.2, we get easily the right-to-left direction.

For the left-to-right direction, we use action testers. We show that $\mathcal{R} = \{([\![P]\!], [\![Q]\!]) \mid P \cong_{\pi_{\mathrm{DiLL}}} Q\}$ is a weak bisimulation.

- Assume that $[\![P]\!] \longrightarrow \sigma$, then we directly apply adequacy and the definition of $\cong_{\pi_{\mathrm{DiLL}}}$ to conclude.
- Assume that $[\![P]\!] \xrightarrow{e^-} \sigma$, then we can conclude by receptivity.
- The interesting case is when $[\![P]\!] \xrightarrow{(a,v)^+} \sigma$. There are two cases:
  - If $v = \ell$, then $P \equiv a \oplus \ell[\vec{x}].\, P_0$. Then, by definition of $\cong_{\pi_{\mathrm{DiLL}}}$, we must have $Q \equiv a \oplus \ell[\vec{x}].\, Q_0$. Then we consider the context:

$$C[] := (\nu ab)[] \mid b \,\&\, \ell(\vec{y})..[\vec{x} \leftrightarrow \vec{y}].$$

  We have $C[P] \equiv P_0$ and $C[Q] \equiv Q_0$. Hence, because $\cong_{\pi_{\mathrm{DiLL}}}$ is a congruence, we have that $P_0 \cong_{\pi_{\mathrm{DiLL}}} Q_0$ as desired.

– If $v = \mathtt{Req}$, then the reasoning is the same.

□

D.3 Second-order full abstraction

We now show Theorem 6.7.

Lemma D.1. *Consider a base type σ and $x : \sigma, \Delta \vdash M, N : \tau$. Then $M \cong_{ML_{\|}} N$ if and only if for every value v of type σ, $M[x := v] \cong_{ML_{\|}} N[x := v]$*

Proof. Left-to-right is by definition of $\cong_{\mathsf{ML}_{\|}}$. Right-to-left, it is easy to see that a context can only use M and N after it has fed a value for x. □

Theorem 6.7 (Full Abstraction). *Consider two terms M, N well-typed on a second-order interface $\Delta \vdash \sigma$. Then $M \cong_{ML_{\|}} N$ if and only if $[\![M]\!] \approx [\![N]\!]$.*

Proof. We only need to show the left-to-right inclusion. By Lemma D.1, we can assume that Δ only contains function types, ie. $\Delta = f_1 : \sigma_1 \to \tau_1, \ldots, \sigma_n \to \tau_n$. We provide a context on $\Delta \vdash \sigma$ that translate function calls into a reference write followed by a reference read. We can without loss of generality suppose that all base types involved are int.

$$\begin{aligned} C[\,] ::= & \mathtt{let}\ \vec{a_i} = \overrightarrow{\mathtt{var}\ 0}\ \mathtt{in} \\ & \mathtt{let}\ \vec{r_i} = \overrightarrow{\mathtt{var}\ 0}\ \mathtt{in} \\ & [\,][f_i := \lambda i.a_i := i; !o_i] \end{aligned}$$

This context creates two references per external function (one to pass the argument value, and one to receive the return) and instanciates f_i with a function that writes the argument, and tries to read back the value.

By assumption, we have that $C[M] \approx_{\mathsf{ML}_{\|}} C[N]$, which in turn implies that $[\![(\!|C[M]|\!)]\!] \approx [\![(\!|C[N]|\!)]\!]$. But it is easy to see that this implies that $[\![M]\!] \approx [\![N]\!]$: each write to a_i corresponds to a call to f_i with the corresponding argument, and each read to r_i correspond to the return value. We conclude by full abstraction. □