Internship report
Concurrent games as event structures

Théophile Bastian, supervised by Glynn Winskel and Pierre Clairambault
Cambridge University

June-July 2016

Abstract

During my internship, I have worked on a *deterministic* game semantics model for a minimalistic *concurrent* language, described using the *games as event structures* formalism. I have proved the *adequacy* of this model with the operational semantics of the language modelled; and I have implemented this model, allowing one to input an expression of the language and getting its representation as Dot graph. This implementation also supports basic operations on event structures, which could be useful to other people working in this domain.

Contents

1 Introduction

Game semantics are a kind of denotational semantics in which a program's behavior is abstracted as a two-players game, in which Player plays for the program and Opponent plays for the environment of the program (the user, the operating system, ...). The execution of a program, in this formalism, is then represented as a succession of moves. For instance, the user pressing a key on the keyboard would be a move of Opponent, to which Player could react by triggering the corresponding action (*eg.* adding the corresponding letter in a text field).

Game semantics emerged mostly with [HO00] and [AJM00], independently establishing a fully-abstract model for PCF using game semantics, while "classic" semantics had failed to provide a fully-abstract, reasonable and satisfying model. But this field mostly gained in notoriety with the development of techniques to capture imperative programming languages constructions, among which references handling [AM96], followed by higher-order references [AHM98], allowing to model languages with side effects; or exception handling [Lai01b]. Since then, the field has been deeply explored, providing a wide range of such constructions in the literature.

A success of game semantics is to provide *compositional* and *syntax-free* semantics. Syntax-free, because representing a program as a strategy on a game totally abstracts it from the original syntax of the programming language, representing only the behavior of a program reacting to its execution environment, which is often desirable in semantics. Compositional, because game semantics are usually defined by induction over the syntax, thus easily composed. For instance, it is worth noting that the application of one term to another is represented as the *composition* of the two strategies.

Concurrency in game semantics. In the continuity of the efforts put forward to model imperative primitives in game semantics, it was natural to focus at some point on modelling concurrency. The problem was first tackled by Laird [Lai01a], introducing game semantics for a λ-calculus with a few additions, as well as a *parallel execution* operator and communication on channels. Ghica and Murawski then simplified Laird's approach, and gave a fully abstract model for a slightly more realistic concurrent programming language with shared memory in [GM04].

However, both of these constructions are based on *interleavings*. That is, they model programs on *tree-like games*, games in which the moves that a player is allowed to play at a given point are represented as a tree (*eg.*, in a state A, Player can play the move x by following an edge of the tree starting from A, thus reaching B and allowing Opponent to play a given set of moves — the outgoing edges of B). The concurrency is then represented as the *interleaving* of all possible sequences of moves, in order to reach a game tree in which every possible "unordered" (*ie.*, that is not enclosed in any kind of synchronisation block, as with semaphores) combination of moves is a valid path.

However, this approach introduces non-determinism in the strategies: if two moves are available to a player, the model states that they make a non-deterministic uniform choice. Yet, even for concurrent programs it is often a desirable property that they should behave consistently with the environment, meaning that they are deterministic up to the choice of the scheduler. Such determinism can be ensured statically, via typing. This idea was explored outside of the game semantics context, for instance by [Rey78], establishing a type-checking system to restrict concurrent programs to deterministic ones. Some recent work makes use of linear logic [CP10] for similar purposes as well. Yet, the interleavings game semantics of these languages remains non-deterministic.

The purpose of this internship was to try to take a first step towards the reunification of those two developments. For that purpose, my objective was to give a *deterministic* game semantics to a linear lambda-calculus enriched with parallel and sequential execution operators, as well as synchronization on channels. In order to model this, I used the games as *event structures* formalism, described later on and introduced in [RW11] by S. Rideau and G. Winskel. Roughly, event structures represent a strategy as a *partial order* on the moves, stating that move x can only be played after move y, which is more flexible than tree-like game approaches. Although a full-abstraction result could not be reached — but is not so far away —, I have proved the *adequacy* of the operational and denotational semantics, and have obtained an implementation of the (denotational) game semantics, that is, code that translates a term of the language into its corresponding strategy.

2 A linear λ-calculus with concurrency primitives: $\lambda\mathcal{L}$CCS

The language on which my internship was focused was meant to be simple, easy to parse and easy to work on both in theory and on the implementation. It should of course include concurrency primitives. For these reasons, we chose to consider a variant of CCS [Mil80] — a simple standard language including parallel and sequential execution primitives, as well as synchronization of processes through *channels*

—, lifted up to the higher order through a λ-calculus. The language was then restricted to a *linear* one — that is, each identifier declared must be referred to exactly once —, partly to keep the model simple, partly to meet the determinism requirements through the banning of interference.

2.1 A linear variant of CCS : $\mathcal{L}$CCS

The variant of CCS we chose to use has two base types: *processes* ($\mathbb{P}$) and *channels* ($\mathbb{C}$). It has two base processes, 0 (failure) and 1 (success), although a process can be considered "failed" without reducing to 0 (in case of deadlock).

Terms		Types	
$t, u, \ldots ::= 1$	(success)	$A, B, \ldots ::= \mathbb{P}$	(process)
$\mid 0$	(error)	$\mid \mathbb{C}$	(channel)
$\mid t \parallel u$	(parallel)		
$\mid t \cdot u$	(sequential)		
$\mid (\nu a)t$	(new channel)		

Figure 1: $\mathcal{L}$CCS terms and types

The syntax is pretty straightforward to understand: 0 and 1 are base processes; $\parallel$ executes in parallel its two operands; $\cdot$ executes sequentially its two operands (or synchronizes on a channel if its left-hand operand is a channel); (νa) creates two new channels, a and $\bar{a}$, on which two processes can be synchronized. Here, the "synchronization" simply means that a call to the channel is blocking until its dual channel has been called as well.

The language is simply typed as in figure 2. Note that binary operators split their environment between their two operands, ensuring that each identifier is used at most once, and that no rules (in particular the axiom rules) "forget" any part of the environment, ensuring that each identifier is used at least once. For instance, in $\Gamma = [p : \mathbb{P}, q : \mathbb{P}]$, 0 cannot be typed (*ie.* $\overline{\Gamma \vdash 0 : \mathbb{P}}$ is not a valid rule).

$$\frac{}{\vdash 0 : \mathbb{P}}(Ax_0) \qquad \frac{}{\vdash 1 : \mathbb{P}}(Ax_1) \qquad \frac{}{t : A \vdash t : A}(Ax) \qquad \frac{\Gamma, a : \mathbb{C}, \bar{a} : \mathbb{C} \vdash P : \mathbb{P}}{\Gamma \vdash (\nu a)P : \mathbb{P}}(\nu)$$

$$\frac{\Gamma \vdash P : \mathbb{P} \quad \Delta \vdash Q : \mathbb{P}}{\Gamma, \Delta \vdash P \parallel Q : \mathbb{P}}(\parallel) \qquad \frac{\Gamma \vdash P : \mathbb{P} \quad \Delta \vdash Q : \mathbb{P}}{\Gamma, \Delta \vdash P \cdot Q : \mathbb{P}}(\cdot_{\mathbb{P}}) \qquad \frac{\Gamma \vdash P : \mathbb{P}}{\Gamma, a : \mathbb{C} \vdash a \cdot P : \mathbb{P}}(\cdot_{\mathbb{C}})$$

Figure 2: $\mathcal{L}$CCS typing rules

We also equip this language with operational semantics, in the form of a labeled transition system (LTS), as described in figure 3, where a denotes a channel and x denotes any possible label.

$$\frac{}{a \cdot P \xrightarrow{a} P} \qquad \frac{}{1 \parallel P \xrightarrow{\tau} P} \qquad \frac{}{1 \cdot P \xrightarrow{\tau} P} \qquad \frac{P \xrightarrow{\tau_c} Q}{(\nu a)P \xrightarrow{\tau} Q}(c \in \{a, \bar{a}\}) \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau_a} P' \parallel Q'}$$

$$\frac{P \xrightarrow{x} P'}{P \parallel Q \xrightarrow{x} P' \parallel Q} \qquad \frac{Q \xrightarrow{x} Q'}{P \parallel Q \xrightarrow{x} P \parallel Q'} \qquad \frac{P \xrightarrow{x} P'}{P \cdot Q \xrightarrow{x} P' \cdot Q} \qquad \frac{P \xrightarrow{x} P'}{(\nu a)P \xrightarrow{x} (\nu a)P'}(x \notin \{a, \tau_a\})$$

Figure 3: $\mathcal{L}$CCS operational semantics

We consider that a term P *converges* whenever $P \xrightarrow{\tau}{}^{*} 1$, and we write $P \Downarrow$.

The τ_a reduction scheme may sound a bit unusual. It is, however, necessary. Consider the reduction of $(\nu a)(a \cdot 1 \parallel \bar{a} \cdot 1)$: the inner term τ_a-reduces to 1, thus allowing the whole term to reduce to 1; but

if we replaced that τ_a with a τ, the whole term would reduce to $(\nu a)1$, which has no valid type since a and $\bar{a}$ are not consumed (linearity). Our semantics would then not satisfy subject reduction for τ-reductions.

2.2 Lifting to the higher order: linear λ-calculus

In order to reach the studied language, $\lambda\mathcal{L}$CCS, we have to lift up $\mathcal{L}$CCS to a λ-calculus. To do so, we add to the language the constructions of figure 4, which are basically the usual λ-calculus constructions slightly transformed to be linear (which is mostly reflected by the typing rules).

Terms		Types	
$t, u, \ldots ::= x \in \mathbb{V}$	(variable)	$A, B, \ldots ::= A \multimap B$	(linear arrow)
$\mid t\ u$	(application)	$\mid \mathbb{P} \mid \mathbb{C}$	($\mathcal{L}$CCS)
$\mid \lambda x^A \cdot t$	(abstraction)		
$\mid \mathcal{L}$CCS *constructions*			

Figure 4: Linear λ-calculus terms and types

To keep the language simple and ease the implementation, the λ-abstractions are annotated with the type of their abstracted variable. The usual $\to$ symbol was also changed to $\multimap$, to clearly remind that the terms are *linear*.

In order to enforce the linearity, the only typing rules of the usual λ-calculus that have to be changed are the (Ax) and (App) presented in figure 5. The (Abs) rule is the usual one.

$$\frac{}{x : A \vdash x : A}(Ax) \qquad \frac{\Gamma \vdash t : A \multimap B \quad \Delta \vdash u : A}{\Gamma, \Delta \vdash t\ u : B}(App) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A \cdot t : A \multimap B}(Abs)$$

Figure 5: Linear λ-calculus typing rules

The linearity is here guaranteed: in the (Ax) rule, the environment must be $x : A$ instead of the usual $\Gamma, x : A$, ensuring that each variable is used *at least once*; while the environment split in the binary operators' rules ensures that each variable is used *at most once* (implicitly, $\Gamma \cap \Delta = \varnothing$).

To lift the operational semantics to $\lambda\mathcal{L}$CCS, we only need to add one rule:

$$\frac{P \longrightarrow_\beta P'}{P \xrightarrow{\tau} P'}$$

2.3 Examples

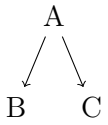
- Simple channel usage: $T_1 \coloneqq (\nu a)\,(a \cdot 1 \parallel \bar{a} \cdot 1)$. This term converges: $(a \cdot 1 \parallel \bar{a} \cdot 1) \xrightarrow{\tau_a} 1 \parallel 1$, thus $T_1 \xrightarrow{\tau} (1 \parallel 1) \xrightarrow{\tau} 1$.
- Deadlock: $T_2 \coloneqq (\nu a)\,(a \cdot \bar{a} \cdot 1)$. This term does not reduce at all: no reduction is possible under the ν.
- Simple function call: $T_3 \coloneqq ((\lambda x^P \cdot x)1) \parallel 1$ reduces to $1 \parallel 1$.
- Channel passing: $T_4 \coloneqq (\nu f)\,(\nu g)\,\big(f \cdot 1 \parallel \big(\big(\lambda a^C \cdot \lambda b^C \cdot \lambda c^C \cdot ((a \cdot 1) \cdot (b \cdot 1)) \parallel (c \cdot 1)\big)\ \bar{f}\ \bar{g}\ g\big)\big)$, which β-reduces (and thus τ-reduces) to $(\nu f)\,(\nu g)\,\big((f \cdot 1) \parallel \big(\big(\bar{f} \cdot 1\big) \cdot (\bar{g} \cdot 1)\big) \parallel (g \cdot 1)\big)\big)$. Note that the function has no idea whether two channels are dual or not, that is, its declaration ignores duality relations between its parameters. In practice, it means that no synchronization can happen before the term is β-reduced.

3 A games model

Our goal is now to give a deterministic games model for the above language. For that purpose, we will use *event structures*, providing an alternative formalism to the often-used tree-like games.

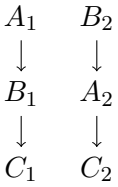
3.1 The event structures framework

The traditional approach to concurrent games is to represent them as *tree-like games*. If the considered game consists in three moves, namely A, B and C, where A can be played by Opponent and either one of the others by Player *after* Opponent has played A, that means that the states of the game will be ϵ, A, $A \cdot B$ and $A \cdot C$, which corresponds to the game tree



This can of course be used to describe much larger games, and is often useful to reason concurrently. The different configurations of the game that can be reached are quite easily read: one only has to concatenate the events found along path starting at the root of the tree.

But it also has the major drawback of growing exponentially in size: let us consider a game in which Opponent must play A and B in no particular order before Player can play C. The corresponding tree-like game would be



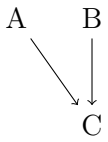
This goes even worse with less structure: since there are $n!$ permutations for n elements, the tree can grow way larger.

The previous example also points out a kind of *obfuscation* of the causal histories: reading the diagram above does not make it obvious to the reader that A and B must be played before C.

This problem can be solved by using *event structures* as a formalism to describe such games [RW11]. Informally, an event structure is a partial order $\leq$ on *events* (here, the game's moves), alongside with a *consistency* relation.

The purpose of the consistency relation is to describe non-determinism, in which we are not interested here, since we seek a deterministic model: in all the following constructions, I will omit the consistency set. The original constructions including it can be found for instance in [CCRW16, Win86].

The partial order $e_1 \leq e_2$ means that e_1 must have been played before e_2 can be played. For instance, the Hasse diagram of the previous game would look like



3.1.1 Event structures

Definition (*event structure*)

An *event structure* [Win86] is a poset $(E, \leq_E)$, where E is a set of *events* and $\leq_E$ is a partial order on E such that for all $e \in E$, $[e] \coloneqq \{e' \in E \mid e' \leq_E e\}$ is finite.

The partial order $\leq_E$ naturally induces a binary relation $\rightarrowtriangle$ over E that is defined as the transitive reduction of $\leq_E$, *ie.* the minimal subset of $\leq_E$ such that the transitive closures of $\leq_E$ and $\rightarrowtriangle$ are the same.

In this context, the right intuition of event structures is a set of events that can occur, the players' moves, alongside with a partial order stating that a given move cannot occur before another move.

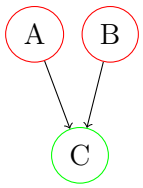
Event structures are often represented as a directed acyclic graph (DAG) where the vertices are the elements of E and the edges are the transitive reduction of $\leq_E$ (*ie.* $\rightarrowtriangle_E$).

Definition (*event structure with polarities*)

An *event structure with polarities* (*ESP*) is an event structure $(E, \leq_E, \rho)$, where $\rho : E \to \{+, -\}$ is a function associating a *polarity* to each event.

In order to model games, this is used to represent whether a move is to be played by Player or Opponent. To represent polarities, we will often use colors instead of + and − signs: a red-circled event will have a negative polarity, *ie.* will be played by Opponent, while a green-circled one will have a positive polarity.

The ESP of the previous example would then be



Definition (*configuration*)

A *configuration* of an ESP A is a finite subset $X \subseteq A$ that is *down-closed*, *ie.*

$$\forall x \in X, \forall e \in A, e \leq_A x \implies e \in X.$$

We write $\mathscr{C}(A)$ the set of configurations of A.

A configuration can thus be seen as a valid state of the game. The set $\mathscr{C}(A)$ plays a major role in definitions and proofs on games and strategies.

Notation

For $x, y \in \mathscr{C}(A)$, $x \overset{e}{\multimap} y$ states that $y = x \sqcup \{e\}$ (and that both are valid configurations), where $\sqcup$ is used to mean that the standard union ($\cup$) is disjoint. It is also possible to write $x \overset{e}{\multimap}$, stating that $x \sqcup \{e\} \in \mathscr{C}(A)$, or $x \multimap y$.

3.1.2 Concurrent games

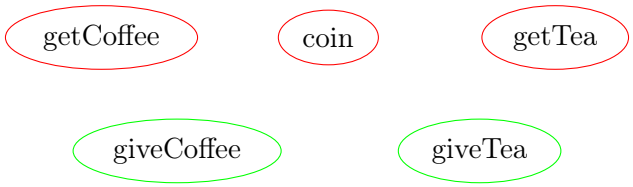
Definition (*game*)

A *game* A is an event structure with polarities.
The dual game $A^\perp$ is the game A where all the polarities in ρ have been reversed.

For instance, one could imagine a game modeling the user interface of a coffee machine: Player is the coffee machine, while Opponent is a user coming to buy a drink.

Example (*coffee machine*)

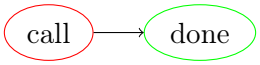
A game describing a coffee machine could be the following one:



Note that there are no edges at all. Indeed, we are here describing the *game*, that is, giving a structure on which we can model any software that would run on the hardware of the coffee machine. Nothing is hardwired that would make it mandatory to insert a coin before getting a coffee: the *software* decides that, it is thus up to the *strategy* — of which we will talk later on — to impose such constraints.

Example (*process game*)

We can represent a process by the following game:



The "call" event will be triggered by Opponent (the system) when the process is started, and Player will play "done" when the process has finished, if it ever does. The relation `call` $\leq$ `done` means that a process cannot finish *before* it is called: unlike what happened in the previous example, it is here a "hardwired" relation that the software cannot bypass.

Definition (*pre-strategy*)

A *pre-strategy* on the game A, written $\sigma : A$, is an ESP such that

(i) $\sigma \subseteq A$ (inclusion over set of events, not including the order);

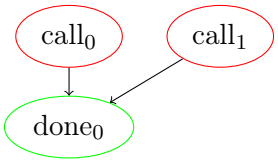
(ii) $\mathscr{C}(\sigma) \subseteq \mathscr{C}(A)$;

(iii) $\forall s \in \sigma, \rho_A(s) = \rho_\sigma(s)$

In particular, (ii) imposes that $\leq_A$ restrained to σ is included in $\leq_\sigma$.

Example (*processes, cont.*)

A possible *pre-strategy* for the game consisting in two processes put side by side (in which the game's events are annotated with a number to distinguish the elements of the two processes) would be



This pre-strategy is valid: it is a subset of the game that does not include call_1, but it does include both call_0 and done_0 and inherits the game's partial order.

This would describe two processes working in parallel. The process 0 waits before the process 1 is called to terminate, and the process 1 never returns. Assuming that `called` is an initially false boolean shared between the two processes, this could for instance be written

Process 0

```c
int main() {
    while(!called) {}
    return 0;
}
```

Process 1

```c
int main() {
    called=true;
    while(true) {}
    // never returns.
}
```

But as it is defined, a pre-strategy does not exactly capture what we expect of a *strategy*: it is too expressive. For instance, the relation $\text{call}_0 \leq \text{call}_1$ on a variant of the above strategy would be allowed, stating that the operating system cannot decide to start the process 1 before the process 0. It is not up to the program to decide that, this strategy is thus unrealistic. We then have to restrict pre-strategies to *strategies*:

Definition (*strategy*)

A *strategy* is a pre-strategy $\sigma : A$ that "behaves well", *ie.* that is

(i) *receptive*: $\forall x \in \mathscr{C}(\sigma), x \sqcup \{e\} \in \mathscr{C}(A) \wedge \rho(e) = \ominus \implies x \sqcup \{e\} \in \mathscr{C}(\sigma)$

(ii) *courteous*: $\forall e \rightarrowtriangle_\sigma e' \in \sigma,\ (\rho(e), \rho(e')) \neq (-,+) \implies e \rightarrowtriangle_A e'$.

(i) captures the idea that we cannot prevent Opponent from playing one of its moves. Indeed, not including an event in a strategy means that this event *will not* be played. It is unreasonable to consider that a strategy could forbid Opponent to play a given move, unless the game itself forbids that as well.

(ii) states that unless a dependency relation is imposed by the games' rules, one can only make one of its moves depend on an Opponent move, *ie.* every direct arrow in the partial order that is not inherited from the game should be $\ominus \rightarrowtriangle \oplus$. Clearly, it is unreasonable to consider an arrow $\circledast \rightarrowtriangle \ominus$, which would mean forcing Opponent to wait for a move (either from Player or Opponent) before playing their move; but $\oplus \rightarrowtriangle \oplus$ is also unreasonable, since we're working in a concurrent context. Intuitively, one could think that when playing e then e', it is undefined whether Opponent will receive e then e' or e' then e.

3.1.3 Operations on games and strategies

In order to manipulate strategies and define them by induction over the syntax, the following operations will be extensively used. It may also be worth noting that in the original formalism [CCRW16], games, strategies and maps between them form a bicategory in which these operations play special roles.

In this whole section, unless stated otherwise, E and F denotes ESPs; A, B and C denotes games; $\sigma : A$ and $\tau : B$ denotes strategies.

Definition (*parallel composition*)

The *parallel composition* $E \parallel F$ of two ESPs is an ESP whose events are $(\{0\} \times E) \sqcup (\{1\} \times F)$ (the disjoint tagged union of the events of E and F), and whose partial order is $\leq_E$ on E and $\leq_F$ on F, with no relation between elements of E and F.

One can then naturally expand this definition to games (by preserving polarities) and to strategies.

In the example before, when talking of "two processes side by side", we actually referred formally to the parallel composition of two processes, in which we took the liberty of renaming the events for more clarity (which we will often do).

Given two strategies on dual games A and $A^\perp$, it is natural and interesting to compute their *interaction*, that is, "what will happen if one strategy plays against the other".

Definition (*closed interaction*)

Given two strategies $\sigma : A$ and $\tau : A^\perp$, their *interaction* $\sigma \wedge \tau$ is the ESP $\sigma \cap \tau \subseteq A$ from which causal loops have been removed.

More precisely, $\sigma \cap \tau$ is a set adjoined with a *preorder* $(\leq_\sigma \cup \leq_\tau)^*$ (transitive closure) that may not respect antisymmetry, that is, may have causal loops. The event structure $\sigma \wedge \tau$ is then obtained by removing all the elements contained in such loops from $\sigma \cap \tau$, yielding a partial order.

This construction is a simplified version of the analogous one from [CCRW16] (the pullback), taking advantage of the fact that our event structures are deterministic — that is, without a consistency set.

This indeed captures what we wanted: $\sigma \wedge \tau$ contains the moves that both σ and τ are ready to play, including both orders, except for the events that can never be played because of a "deadlock" (*ie.* a causal loop).

We might now try to generalize that to an *open* case, where both strategies don't play on the same games, but only have a common part. Our goal here is to *compose* strategies: indeed, a strategy on $A^\perp \parallel B$ can be seen as a strategy *from* A *to* B, playing as Opponent on a board A and as Player on a board B. This somehow looks like a function, that could be composed with another strategy on $B^\perp \parallel C$ (as one would compose two mathematical functions f and g into $g \circ f$).

Definition (*compositional interaction*)

Given two strategies $\sigma : A^\perp \parallel B$ and $\tau : B^\perp \parallel C$, their *compositional interaction* $\tau \circledast \sigma$ is an event structure defined as $(\sigma \parallel C^\perp) \wedge (A \parallel \tau)$, where A and $C^\perp$ are seen as strategies.

The idea is to put in correspondence the "middle" states (those of B) while adding "neutral" states for A and C, which gives us two strategies playing on the same game (if we ignore polarities), $A \parallel B \parallel C$.

Here, we define $\tau \circledast \sigma$ as an *event structure* (*ie.*, without polarities): indeed, the two strategies disagree on the polarities of the middle part. Alternatively, it can be seen as an ESP with a polarity function over $\{+, -, ?\}$.

From this point, the notion of composition we sought is only a matter of "hiding" the middle part:

Definition (*strategies composition*)

Given two strategies $\sigma : A^\perp \parallel B$ and $\tau : B^\perp \parallel C$, their *composition* $\tau \odot \sigma$ is the ESP $(\tau \circledast \sigma) \cap (A^\perp \parallel C)$, on which the partial order is the restriction of $\leq_{\tau \circledast \sigma}$ and the polarities those of σ and τ.

It is then useful to consider an identity strategy *wrt.* the composition operator. This identity is called the *copycat* strategy: on a game A, the copycat strategy playing on $A^\perp \parallel A$ replicates the moves the other player from each board on the other.

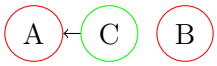
Definition (*copycat*)

The *copycat strategy* of a game A, $\mathfrak{cc}_A$, is the strategy on the game $A^\perp \parallel A$ whose events are $A^\perp \parallel A$ wholly, on which the order is the transitive closure of $\leq_{A^\perp \parallel A} \cup \{(1-i, e) \leq (i, e) | e \in A \ \& \ \rho((i, e)) = \oplus\}$.

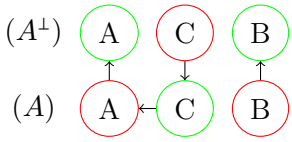
The copycat strategy of a game is indeed an identity for the composition of *strategies*. In fact, it even holds that for a *pre*-strategy $\sigma : A$, σ is a strategy $\iff \mathfrak{cc}_A \odot \sigma = \sigma$ [RW11].

Example (*copycat*)

If we consider the following game A



its copycat strategy $\mathbb{c}_A$ is



Note that the edge $C \to A$ in the upper row is no longer needed, since it can be obtained transitively and we only represent the transitive reduction of the partial order.

3.2 Interpretation of $\lambda\mathcal{L}$CCS

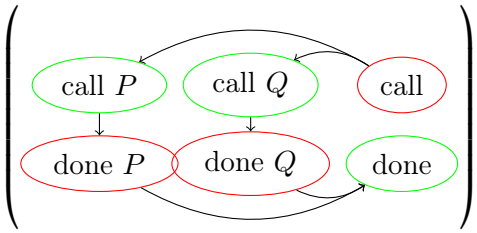
We can now equip $\lambda\mathcal{L}$CCS with denotational semantics, interpreting the language as strategies as defined in figure 6.

$$[\![x^A]\!] \coloneqq \mathbb{c}_{[\![A]\!]} \qquad [\![A \multimap B]\!] \coloneqq [\![A]\!]^\perp \parallel [\![B]\!]$$

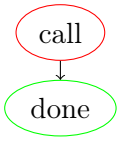
$$[\![t^{A\multimap B}\ u^A]\!] \coloneqq \mathbb{c}_{A\multimap B} \odot ([\![t]\!] \parallel [\![u]\!])$$

$$[\![\lambda x^A \cdot t]\!] \coloneqq [\![t]\!]$$

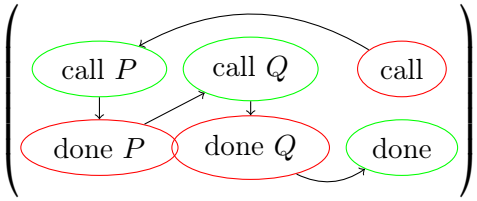
$$[\![P \parallel Q]\!] \coloneqq$$



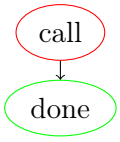
$$\odot ([\![P]\!] \parallel [\![Q]\!]) \qquad [\![\mathbb{P}]\!] = [\![\mathbb{C}]\!] \coloneqq$$



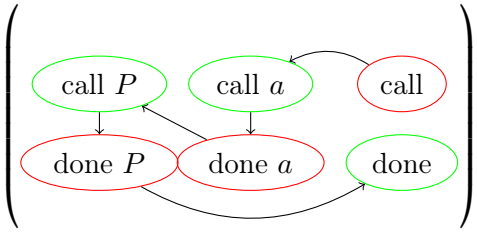
$$[\![P \cdot Q]\!] \coloneqq$$



$$\odot ([\![P]\!] \parallel [\![Q]\!]) \qquad [\![1]\!] \coloneqq$$

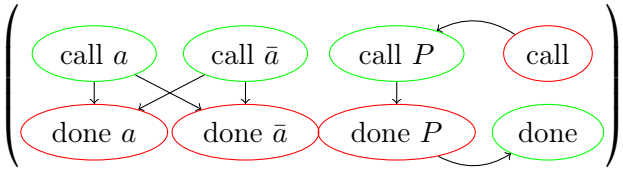


$$[\![(a : \mathbb{C}) \cdot P]\!] \coloneqq$$



$$\odot ([\![P]\!] \parallel [\![a]\!]) \qquad [\![0]\!] \coloneqq$$

$$[\![(\nu a)P]\!] \coloneqq$$



$$\odot [\![P]\!]$$

Figure 6: $\lambda\mathcal{L}$CCS interpretation as strategies

In the representation above, the drawn strategies are organized in columns: for instance, the strategy involved in $[\![P \parallel Q]\!]$ has type $P^\perp \parallel P^\perp \parallel P$. Each column of events stands for one of those

games, in this order.

The $[\![-]\!]$ operator always sends a term $x_1 : A_1, \ldots, x_p : A_p \vdash t : B$ to a strategy $[\![t]\!] : [\![A_1]\!]^\perp \parallel \ldots \parallel [\![A_p]\!]^\perp \parallel [\![B]\!]$. For brevity purposes, the associativity and commutativity steps — up to isomorphism — will be kept implicit here, although those are handled very formally in the implementation.

A lot of these interpretations is what was expected: $\mathbb{P}$ has the same interpretation as presented before, and that of $\mathbb{C}$ is set to the same, 1 is interpreted as the game playing *done* while 0 does not, the arrow is the type introduced for functions.

A variable is represented by a *copycat* of its type, to keep a version of the variable in the environment, reflecting the judgement $x : A \vdash x : A$. Same goes for the abstraction: the typing rule states that given a term verifying $\Gamma, x : A \vdash t : B$, we should obtain a term verifying $\Gamma \vdash \lambda x^A \cdot t : A \multimap B$. The only thing we have to do is to "move" x from the environment to the term, which is transcribed by the associativity of $\parallel$. In the application, one can notice that the copycat wires up perfectly $[\![t]\!] \parallel [\![u]\!]$: the version of x from the environment of t is connected to u, and the output of t is connected to the output of the term.

3.3 Adequacy

We will now describe the main steps of the proof of the major result of this study, the *adequacy* of the game semantics.

Theorem (*adequacy*)

The previous interpretation is *adequate* with respect to the operational semantics, that is

$$\forall P \ st. \ (\vdash P : \mathbb{P}), \ (P \Downarrow) \iff ([\![P]\!] = [\![1]\!])$$

In order to prove the theorem, a few intermediary definitions and results are required.

Definition (*evaluation in a context*)

For l a list of channels and P a term, the evaluation of P in the context l, $[\![P]\!]_l$, is defined by induction by $[\![P]\!]_{[]} \coloneqq [\![P]\!]$ and $[\![P]\!]_{h::t} \coloneqq [\![(\nu h)P]\!]_t$.

Definition (*valid contexts*)

The *valid contexts* for a reduction $P \xrightarrow{x} Q$, $\mathcal{L}_{P \xrightarrow{x} Q}$, is the set of (ordered) lists of channels l such that

(i) $\forall a : \mathbb{C} \in \mathrm{fv}(P) \ st. \ \bar{a} \in \mathrm{fv}(P), \ a \in l$ or $\bar{a} \in l$;

(ii) if $x = a : \mathbb{C}$, $a \notin l$;

(iii) $\forall a : \mathbb{C}, \ a \in l \implies \bar{a} \notin l$.

where $\mathrm{fv}(P)$ denotes the set of free variables of P, defined as usual by induction over the syntax as the set of variables unbound in the term.

Lemma

For all $a : \mathbb{C}$, $P, Q, R : \mathbb{P}$, the following properties hold:

(i) $(\nu a)\,(P \parallel Q) = ((\nu a)\,P) \parallel Q$ when $a, \bar{a} \notin \mathrm{fv}\,Q$;

(ii) $(\nu a)\,(P \parallel Q) = P \parallel ((\nu a)\,Q)$ when $a, \bar{a} \notin \mathrm{fv}\,P$;

(iii) $(\nu a)\,(P \cdot Q) = ((\nu a)\,P) \cdot Q$ when $a, \bar{a} \notin \mathrm{fv}\,Q$;

(iv) $[\![\ ]\!]_l$ is associative, that is, for all l, $[\![(A \parallel B) \parallel C]\!]_l = [\![A \parallel (B \parallel C)]\!]_l$.

The previous lemma's proof mostly consists in technical, formal reasoning on event structures, but it is essentially intuitive.

The theorem is mostly a consequence of the following lemma:

Lemma

$\forall P \xrightarrow{x} Q,\ \forall l \in \mathcal{L}_{P \xrightarrow{x} Q}$,

(i) if $x = \tau$, then $[\![P]\!]_l = [\![Q]\!]_l$;

(ii) if $x = a : \mathbb{C}$, then $[\![P]\!]_l = [\![a \cdot Q]\!]_l$;

(iii) if $x = \tau_a$ $(a : \mathbb{C})$, then $[\![P]\!]_{a::l} = [\![Q]\!]_l$.

Proof. We prove this by induction over the rules of the operational semantics of $\lambda\mathcal{L}$CCS. Most of the cases are straightforward, thus, we will only sketch the proof for a few of those cases.

- The basic rules (for $\parallel, \cdot, \dots$) are working thanks to the previous lemma.
- $\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau_a} P' \parallel Q'}$: for all l, by (ii), $[\![P]\!]_l = [\![a \cdot P']\!]_l$ and $[\![Q]\!] = [\![\bar{a} \cdot Q']\!]$, thus $[\![(\nu a)(P \parallel Q)]\!]_l = [\![\,(\nu a)\,(a \cdot P' \parallel \bar{a} \cdot Q')]\!]$ (inline replacement of terms, permitted by the previous lemma), thus $[\![\,(\nu a)\,(P \parallel Q)]\!]_l = [\![P' \parallel Q']\!]_l$.
- $\dfrac{P \xrightarrow{\tau_c} Q}{(\nu a)\,P \xrightarrow{\tau} Q}(c \in \{a, \bar{a}\})$ clearly works thanks to (iii).

□

Proof: theorem. Forwards implication: $(P \Downarrow) \implies ([\![P]\!] = [\![1]\!])$. Proof by induction over the derivation of $P \xrightarrow{\tau}{}^{*} 1$, by iterating the previous lemma.

Backwards implication: $([\![P]\!] = [\![1]\!]) \implies (P \Downarrow)$. We prove its contrapositive judgement, $P \neq 1 \,\&\, [\![P]\!] = [\![1]\!] \implies \exists Q : P \xrightarrow{\tau} Q$ by induction over the syntax of $\mathcal{L}$CCS (*wlog.*, we can assume that $P \not\rightarrow_\beta$, because we can always do every β-reduction before any other τ-reduction, and because the β-reduced term corresponding to a counter-example is a counter-example as well; thus the terms are in $\mathcal{L}$CCS): for each syntactic construction, we prove that under the induction hypotheses, there is such a Q. □

This proves the adequacy of our semantics, giving some credit to the game semantics we provided: indeed, in order to decide whether a term converges, we can compute its associated strategy and check whether it is $[\![1]\!]$.

4 Implementation of deterministic concurrent games

 Try online[1]  Github repository[2]

One of the goals of my internship was also to implement the operations on games and strategies described in §3.1.3, and to use them to provide a convenient Dot representation of the operational semantics of $\lambda\mathcal{L}$CCS described in §3.2.

4.1 Structures

The implementation aims to stay as close as possible to the mathematical model, while still providing quite efficient operations.

As we do not handle non-determinism, an event structure can be easily represented as a DAG in memory. The actual representation that was chosen is a set of nodes, each containing (as well as a few other information) a list of incoming and outgoing edges.

4.2 Generic operations

The software — apart from a few convenience functions used to input and output games and strategies, as well as the $\lambda\mathcal{L}$CCS handling — mostly consists in graph handling functions (mostly used internally) and the implementation of the operations on games and strategies previously described.

This allows to compute the result of any operation on a deterministic strategy, and is modular enough to make it possible to implement non-determinism on the top of it later on (even without having to understand the whole codebase). Those operations can be used directly from the OCaml toplevel, conveniently initialized with the correct modules loaded with `make toplevel`; but it is mostly intended to be a backend for higher level interfaces, such as the $\lambda\mathcal{L}$CCS interface.

4.3 Modelling $\lambda\mathcal{L}$CCS

The modelling of $\lambda\mathcal{L}$CCS required to implement a lexer/parser for the language and a function transforming $\lambda\mathcal{L}$CCS terms into strategies, as well as a rendering backend, displaying a strategy as a Dot graph. This could then just be plugged into an HTML/Javascript frontend using `js_of_ocaml`; this frontend is linked above in the document.

The major difficulty came from the necessity to massively reorder the sub-terms of the strategies on the go: indeed, in order to know how to compose two strategies σ and τ, the implementation keeps track of the parallel compositions that were taken to get both strategies. For instance, if σ was obtained by putting in parallel strategies so that the game is $(A \parallel B^\perp) \parallel C^\perp$, and τ was obtained the same way, reaching a game $(B \parallel C) \parallel D$, the implementation would refuse to compute $\tau \odot \sigma$, because it would try to match games $C^\perp$ and $(B \parallel C)$.

The theoretical construction extensively uses the associativity (up to isomorphism) of $\parallel$, but in the code, each use of the associativity must be explicit, leading to a large amount of code.

5 Conclusion

During this internship, I have established deterministic game semantics for a simple concurrent language. Although the language is fairly simple, it should not be too hard to lift it to a language closer to real-world programming languages, through the inclusion of the imperative primitives found in the literature.

I also explored the possibility to reach a full-abstraction result. The full-abstraction property states that two terms are *observationally equivalent* if and only if their (denotational) semantics are

[1] https://tobast.fr/l3/demo.html
[2] https://github.com/tobast/cam-strategies/

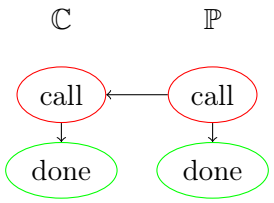
also observationally equivalent, that is, in this context, for all P, Q two $\lambda\mathcal{L}$CCS terms such that $\Gamma \vdash P, Q : A$,

$$[\forall \mathcal{C}[-] \text{ a context } : (\vdash \mathcal{C}[P] : \mathbb{P}),\ \mathcal{C}[P] \Downarrow \iff \mathcal{C}[Q] \Downarrow] \quad \iff \quad (\llbracket P \rrbracket \simeq \llbracket Q \rrbracket)$$

where $\simeq$ denotes the observational equivalence on strategies, that is,

$$(\sigma : A) \simeq (\tau : A) \iff \forall (\alpha : A^{\perp} \parallel \mathbb{P}), \alpha \odot \sigma = \alpha \odot \tau$$

Yet, by lack of time, I had to abandon this path. Indeed, this would have required either to modify $\lambda\mathcal{L}$CCS or to restrict the authorized strategies, because of the following legal strategy, which cannot be expressed as the semantics of a term in $\lambda\mathcal{L}$CCS:



This strategy behaves like a "forget" strategy: its effect is to call one end of a channel, and then to resume the execution of the term without waiting for the other end to be called. If we integrate this operator to the language as $(f\, a)$ for any channel a, this construction can discriminate terms that would not have been discriminated before. For instance $\lambda x^{\mathbb{C} \multimap \mathbb{P}} \cdot (\nu a)\,((xa) \cdot \bar{a} \cdot 1)$ and $\lambda x^{\mathbb{C} \multimap \mathbb{P}} \cdot (\nu a)\,((xa) \cdot \bar{a} \cdot 0)$ can be discriminated by the context $\mathcal{C}[X] = X(\lambda a^{\mathbb{C}} \cdot (fa))$.

References

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on*, pages 334–344. IEEE, 1998.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electronic Notes in Theoretical Computer Science*, 3:2–14, 1996.

[CCRW16] Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. Concurrent games. *arXiv preprint arXiv:1604.04390*, 2016.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, pages 222–236. Springer, 2010.

[GM04] Dan R Ghica and Andrzej S Murawski. Angelic semantics of fine-grained concurrency. In *International Conference on Foundations of Software Science and Computation Structures*, pages 211–225. Springer, 2004.

[HO00] J Martin E Hyland and C-HL Ong. On full abstraction for PCF: I, II, and III. *Information and computation*, 163(2):285–408, 2000.

[Lai01a] James Laird. A game semantics of Idealized CSP. *Electronic Notes in Theoretical Computer Science*, 45:232–257, 2001.

[Lai01b] James Laird. A fully abstract game semantics of local exceptions. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 105–114. IEEE, 2001.

[Mil80] Robin Milner. *A calculus of communicating systems*. 1980.

[Rey78] John C Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM, 1978.

[RW11] Silvain Rideau and Glynn Winskel. Concurrent strategies. In *LICS*, volume 11, pages 409–418, 2011.

[Win86] Glynn Winskel. Event structures. In *Advanced Course on Petri Nets*, pages 325–392. Springer, 1986.