Concurrent Games in Agda

Amy Yin

4th Year Project Report
Cognitive Science
School of Informatics
University of Edinburgh

2024

Abstract

Concurrent games are a formalism used in game semantics, a branch of denotational semantics. This project concerns an Agda mechanisation of concurrent games and strategies based the constructions provided by *Games and Strategies as Event Structures* (Castellan, Clairambault, Rideau & Winskel, 2016), as well as the implementation of an Agda library for event structures, a formalism used in concurrency theory to describe computational processes.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Amy Yin*)

Acknowledgements

I would like to express my sincere gratitude to Dr. Liam O'Connor for his knowledge, guidance and support. Without him, this project would not have been possible. I thank Professor Rob van Glabbeek and Professor Glynn Winskel for their invaluable insights and assistance. I also thank my friends Kate and Ellie for loaning me their cat Stella, who has been an anchor for me in these trying times.

Table of Contents

Chapter 1

Introduction

Games are formal structures that model computational processes in terms of a dialogue between two agents. When applied to formal semantics, games have provided a wealth of important results. There are a number of approaches when it comes to expressing games as formal structures, and the approach this dissertation is concerned with is *concurrent games* — games modelled with *event structures*.

This chapter provides some background on game semantics, a short survey of any efforts towards mechanising semantic games through proof assistants like Agda, and an overview of this project.

1.1 Background

A game, as studied in game semantics, is an interaction abiding by the following structure (Abramsky and McCusker [1999], Hyland [1997]):

- There are two players, O (Opponent) and P (Player), who make *moves* on strictly alternating turns.
- O always moves first, and *P-moves* are made according to a predetermined *strategy* in response to *O-moves*.
- The game may or may not terminate, and depending on the type of game being played, there may not be any notion of “winning”(Clairambault et al. [2012]).

While this definition may seem informal, this really is all there is to games at the highest level of abstraction. The more formal definition that will be introduced in a later chapter is an *event structure* designed to exhibit the behaviour described above.

Outside of formal semantics, games find applications in logic as a technique for determining the truth-value of a sentence in first-order logic (Hintikka and Sandu [1997]), as well as concurrency theory and logic for the determination of structural equivalence (Ehrenfeucht [1961]). To avoid confusion, it may also help to clarify that these games bear little to no relation with the games studied in game theory.

1.1.1 Game semantics

Game semantics is a semantic theory for programming languages that interprets the behaviour of programs in terms of games, where Player P corresponds to the observable behaviour of a program, and Player O represents the computational environment in which the program is executed (Hyland [1997]).

We will now adopt a few examples from Murawski and Tzevelekos [2016] to illustrate how games can be used to describe the behaviour of some simple programs.

Consider a program that returns a constant:

$$\vdash 42\colon \texttt{int}$$

This program can be interpreted as a brief dialogue between O and P, as follows:

O	What is the result?
P	42.

In this very simple game, there are two things worth noting: firstly, the *P-move* of responding with '42' is a strategy predetermined by the nature of the program. Secondly, this strategy is shared by *any* closed program of the shape $\vdash M\colon \texttt{int}$ that evaluates to 42, regardless of what M is. This is because the environment of a program does not get to look under the hood of a program, only having access to the program's outputs.

For a more interesting game, consider the following program that evaluates the successor of an input with type `int`:

$$\vdash \lambda x^{int}.\, x+1\colon \texttt{int} \to \texttt{int}$$

The environment can now call the function repeatedly by supplying it with arguments, which corresponds to call-by-value evaluation. A possible dialogue for this term may look like the following:

O	What is the result?
P	It is a function.
O	What is the result if the argument is 0?
P	1.
O	What is the result if the argument is 42?
P	43.

Again, it is important to note that the body of the function in the program, $x+1$, does not appear explicitly in the *play*.

At this point one may have noticed that in both of the previous examples, O is always the player asking questions while P is always the player providing answers. This is because neither of the two examples above contain *free variables*, corresponding to behaviours undefined in the program. When a free variable shows up, it is up to the environment to provide the program with information about it. For example, we might have the following program:

$$f \; : \; \mathtt{int} \to \mathtt{int} \vdash f(f(0)) + 1$$

Here, f is undefined in the body of the program, so it is the environment's job to supply the results of calling f. For example:

O	What is the result?
P	What is $f(0)$?
O	3.
P	What is $f(3)$?
O	4.
P	5.

It should be possible to extrapolate from the above that more complicated games can be devised for more complicated programs.

Now, *denotational semantics* is the practice of interpreting the semantic meaning of programs in terms of compositional mathematical objects. Games semantics is a type of denotational semantics because games can be understood as one such compositional mathematical object: all we need is a mathematical model of games that permits the composition of two games into a bigger game.

The traditional choice for this mathematical model is trees, where consecutive nodes of a tree are used to represent moves and assigned alternating *polarities* in correspondence to the player they belong to. Trees are then equipped with *justification pointers* to indicate the *enabling* of an answer-move by a corresponding question-move, and useful concepts such as *arenas*, *visibility*, or *innocent strategies* can be subsequently defined. The full abstraction result for PCF (Programming Computable Functions) was achieved simultaneously through two different approaches using trees: One due to Abramsky and McCusker [1999] and the other due to Hyland and Ong [2000].

A new alternative to the tree-based approach uses *event structures* as the underlying formal model for games (Rideau and Winskel [2011]). Broadly speaking, event structures express computational processes as *event occurrences* with causal relations (Winskel [1987]), and the rest of this dissertation will examine the construction of games as event structures in close formal detail.

A crucial difference between tree-based approaches and the event structure-based approach is that the latter can model true concurrency, while the former are limited to interleaving concurrency due to the sequential nature of tree-nodes (Rideau and Winskel [2011]). Therefore, event structure-based games are also called *concurrent games* or *distributed games* (the names are used interchangeably in the literature).

Concurrent games have generated a wealth of results in recent years, including a replication of the full abstraction result for PCF (Castellan et al. [2015]), full abstraction for the quantum lambda calculus (Clairambault and de Visme [2019]), as well as the establishment of winning conditions (Clairambault et al. [2012]) and payoffs in semantic games (Clairambault and Winskel [2013]) with the goal of bringing them closer to the games studied in game theory.

1.2 Project summary

Stated briefly, the aim of this project is to develop a library for concurrent games in Agda, an interactive proof assistant. This entail a partial mechanisation of the constructions developed by [Castellan et al., 2017], up to their definition of concurrent strategies. As all definitions used in this project are sourced from [Castellan et al., 2017], we will refer to it as CCRW from this point onwards.

1.2.1 Motivation

The construction of machine-checked proofs through the use of a proof assistant like Agda provide a guarantee of correctness for the constructions employed by CCRW. To this end, an Agda library containing the definitions related to concurrent games must be implemented. It is our goal to complete this library as a foundation for more ambitious future projects like the mechanisation of concurrent game semantics for a programming language.

Mechanised proofs generally demand a higher level of rigour than pen-and-paper proofs. For example, this project includes a number of proofs on properties such as the associativity of the parallel composition of event structures, while this was entirely left to the intuition of the reader in CCRW. By establishing this additional level of rigor, we improve confidence in CCRW.

Additionally, mechanisation is an open problem in game semantics. The only published piece of work including a mechanisation of semantic games is for *Hyland-Ong games* (Churchill and Laird [2010]) rather than concurrent games. Therefore, our work is novel in targeting the newer approach of concurrent games.

1.2.2 Scope

This project mechanises the constructions necessary for expressing the notion of a *concurrent strategy* in Agda, where strategies are sequences of moves on the semantic games described above. More specifically, mechanisations were developed for:

- Event structures, and operations on event structures including parallel composition, interaction, and mappings.
- Concurrent games (or, event structures with polarities).
- Concurrent *pre-strategies* and operations on them, including interactions and compositions.
- The *copycat pre-strategy*, the identity morphism in $\mathcal{CG}$, the (bi)category of games.
- And finally, strategies on concurrent games that act as morphisms on $\mathcal{CG}$.

While the previous section situates concurrent games in the study of game semantics because they were originally developed for this purpose, this project does not actually use the games and strategies developed as denotations for a programming language due to the immensity of this undertaking — for reference, the development of *Innocent* *game models of untyped λ-calculus* (Ker et al. [2002]) was enough to fill a PhD thesis that does not contain any mechanisation work.

Here we acknowledge that in its current state, the project is incomplete for two reasons: we leave some lemmas unproved through *open holes* in the Agda code, and we employ a postulate in place of implementing a full Agda library for finite sets. We leave the unfinished proofs as future work as we are confident that they can be completed given more time.

The scope of this project also reflects Agda's limited capacity towards the mechanisation of mathematics. The exact nature of these limitations will be addressed in future sections of this report in accordance with aspects of the implementation work from which they arise, as well as the Evaluation chapter.

1.2.3 Contributions

This project makes two primary contributions:

1. *A library for event structures in Agda*: To our knowledge, there is only one existing piece of mechanisation work on event structures and it is completed using the theorem prover formerly known as Coq (V. P. Gladstein [2021]), rather than Agda.

2. *A library for concurrent games in Agda*: As stated previously, the only existing mechanisation of the games used in game semantics also uses Agda, targeting Hyland-Ong games (Churchill and Laird [2010]) rather than concurrent games.

1.2.4 The structure of this report

The bulk of this report is dedicated towards explaining the formal constructions leading up to concurrent strategies in the order they were listed by subsection 1.2.2. Each new definition will be immediately accompanied by its corresponding mechanisation in Agda. After the expositions for concurrent strategies and our mechanisations are complete, there will be a concluding chapter covering the critical evaluation and future directions of our work.

In the interest of space and relevance, a number of supporting lemmas required by our implementation will be omitted from this report. A notable example is the library of operations on finite sets we implement because Agda does not have an existing library for them. The reader is invited to access the Agda source code associated with this project to verify the existence of the omitted proofs.

1.3 A primer on Agda

Agda is a dependently-typed functional language and interactive theorem prover based on Martin-Löf type theory (Norell [2009]) (Martin-Löf and Sambin [1984]). In Agda, programs correspond to proofs: properties of values are encoded as types, such that the inhabitants of the type are proofs testifying to the truth of these properties. Programs in

Agda must be total for the resulting logic to be consistent: programs that crash or fail to terminate are do not constitute valid proofs (Norell [2009]).

Agda is the language of choice for this project because of its suitability towards proofs concerning formal semantics (see Wadler et al. [2022] for a notable example), as well as the author's familiarity with it.

Before beginning our exposition of the project, we provide a brief overview of Agda's language constructs and some of the techniques we employ.

1.3.1 Records

Records are types that offer a convenient way for grouping values together. They are a generalisation of product types with additional features including named fields, constructors, and locally-defined functions.

Most of the mechanised definitions in our implementation make use of records, because formalisms that must satisfy multiple properties can be expressed as records with multiple fields.

1.3.2 Sets

In Agda, the `Set` keyword refers to the type of small types (Norell [2009]). Due to Russell's paradox Agda implements a universe of type hierarchies, e.g. `Set` itself has type `Set₁`, `Set₁` has type `Set₂`, etc. It is important to note that these are types rather than set-theoretic sets.

Relatedly, the absence of of a library for finite sets in Agda prompted us to implement our own, where we define finite sets and operations on them through Agda's built-in lists. Details of this library and the complications resulting from our approach will be covered in the rest of this report.

1.3.3 Predicates

We express predicates as types that take a set of parameters and return `Set`, a commonly used technique in proof mechanisation with Agda. For example, we can define a predicate `Odd : Nat → Set` as an (inductive) indexed datatype like so:

```
data Odd : Nat → Set where
  odd  : Odd (suc (zero))
  odd-plus2 : {n : Nat} → Odd n → Odd (suc (suc n))
```

This code snippet requires all inhabitants of the type `Odd` to satisfy the properties declared in one of its two constructors, in so ensuring that ∀ `x : Nat` for which `Odd x` holds, `x`'s `Odd`-ness is provably true by construction.

1.3.4 Postulates

The `postulate` keyword permits the declaration of a theorem without a proof. The use of postulates is dangerous because one can easily postulate something that is false (e.g. an element of the empty type $\bot$).

A postulate is used with extreme caution in this project because of Agda's lack of a library for finite sets. We will provide additional justifications for its use in section 3.3.3.

1.3.5 Miscellaneous language constructs

Presuming some familiarity with Haskell-like concrete syntax, we address some features of Agda's syntax and libraries that will make it easier to read the code snippets in this report.

1. *Implicit arguments*: In the following type signature, we put curly brackets {} around `A : Set`:

   ```
   example : ∀ {A : Set} → List A → List A
   ```

   Curly brackets indicate an argument is *implicit* — given an explicit argument of type `List A`, Agda is able to infer the value of the implicit argument, offering a useful way of de-cluttering our proofs.

2. *Named fields in records*: Given a `record` called `R` that contains a field `a`, we project the value of `a` from the record by writing `a R`.

3. *Infix operators*: An infix operator can be written in prefix form. This sometimes becomes confusing if a record field name is also involved. Suppose an infix operation _≤_ is defined within a record `R`. In Agda, the application of _≤_ to some `a` and `b` outside of `R` is written:

   ```
   (R ≤ a) b
   ```

   This is equivalent to writing `(_≤_ R a b)` and expresses "`a` is less than or equal to `b`", not the other way around.

Chapter 2

Event Structures

The following chapter introduces event structures, accompanied by an overview of our corresponding mechanisations in Agda.

2.1 Definition

Event structures model computational processes as *event occurrences* with causal relations.

Definition 2.1.1 (CCRW)**.** *(Event structures). An event structure is a triple $\{E, Con, \leq\}$ where E is a set of events partially ordered by the* causal dependency relation $\leq$. *The partial order $\leq$ is finitary, such that:*

$$\{e' | e' \leq e\} \text{ is finite for all } e \in E$$

Con is a nonempty consistency relation *consisting of finite subsets of E, satisfying the following properties:*

- *$\{e\} \in Con$ for all $e \in E$.*
- *$Y \subseteq X \in Con$ implies $Y \in Con$.*
- *$X \in Con$ and $e \leq e' \in X$ implies $X \cup \{e\} \in Con$.*

The configurations *of E, written $\mathcal{C}(E)$, represent the states of an event structure as finite subsets $x \subseteq E$ satisfying:*

- Consistent*: $X \subseteq x$ and X is finite implies $X \in Con$.*
- Down-closed*: $e' \leq e \in x$ implies $e' \in x$.*

Two events e, e' are *concurrent* if the set $\{e, e'\}$ is in Con, and neither e, e' are causally dependent on the other.

2.2 Event structures in Agda

Implementing a library for event structures in Agda entails formalising each segment of the definition provided above:

1. Events,
2. The finite partial order $\leq$, and
3. The consistency relation *Con*, and
4. Event structures themselves and their configurations.

We note that event structures are not finite posets, because despite being endowed with a finite partial order, the set of events belonging to an event structure can be infinite.

2.2.1 Finite partial orders

The causal dependency relation $\leq$ describes the causal relation between the events of an event structure, i.e. for events $a, b \in E$, $a \leq b$ if b can happen because a happened. The causal dependency relation is a finite partial order, which we implement in Agda because they are not provided in Agda's existing poset library.

Definition 2.2.1. *(Finite partial orders). A partial order on a set* E *is a binary relation satisfying:*

- *Reflexivity* ($\forall e \in E,\ e \leq e$),
- *Anti-symmetry* ($\forall e, e' \in E$, *if* $e \leq e'$ *and* $e' \leq e$ *then* $e = e'$),
- *Transitivity* ($\forall m, n, p \in E$, *if* $m \leq n$ *and* $n \leq p$ *then* $m \leq p$).

A finite partial order is a partial order satisfying:

- $\{e' \mid e' \leq e\}$ *is finite for all* $e \in E$. *We call this set the* downclosure *of* e.

The ordering can be represented as the downclosure function. In our implementation, we represent finite sets in Agda through lists where the order of list items and repeating items are ignored. Then, a finite partial order can be mechanised as a function

```
≤-f  : E → List E
```

upon which the $\leq$ relation can be defined through list membership, i.e.

```
e′ ∈ ≤-f e   iff   e′ ≤ e
```

Armed with the observations above, we can define a finite partial order over a set E as a record like so:

```
record FinitePartialOrder (E : Set) : Set where
  field
    ≤-f : E → List E
    ≤-f-refl : ∀ {e : E} → e ∈ (≤-f e)
    ≤-f-antisym : ∀ {e e′ : E} → e ∈ (≤-f e′) → e′ ∈ (≤-f e) → e′ ≡ e
    ≤-f-trans : ∀ {m n p : E} → m ∈ (≤-f n) → n ∈ (≤-f p) → m ∈ (≤-f p)
```

```
_≤_ : E → E → Set
e′ ≤ e = Any (e′ ≡_) (≤-f e)

open FinitePartialOrder
```

The finiteness of `≤-f` is ensured by its codomain being a list, and it must be a partial order because we require any candidate for `≤-f` to satisfy reflexivity, anti-symmetry, and transistivity via the other fields of the record. Finally, we define an infix operator ≤ as syntactic sugar so that we can avoid explicitly referring to the list operations that we are using as set operations.

2.2.2 Consistency relation

The consistency relation on event structures specifies finite sets of events in E that can happen concurrently. An equivalent definition can be given in terms of *conflicting* events, but we stick to consistent events here for simplicity.

We mechanise the consistency relation for a given finite partial order as a predicate `Con : E → Set` that satisfies the three properties of *Con* in Definition 2.1.1, plus an additional requirement stating the empty set is consistent. We note our continued use of lists as representations of finite sets.

```
record Consistency {E : Set} (con-fpo : FinitePartialOrder E) : Set₁ where
  field
    con-f : List E → Set
    con-1 : ∀ { e : E } → con-f [ e ]
    con-2 : ∀ { xs ys : List E } → con-f xs → ys ⊆ xs → con-f ys
    con-3 : ∀ { e e′ : E } → (es : List E)
    → con-f es → e ∈ es → (con-fpo ≤ e) e′ → con-f (e′  es)
    con-empty : con-f []

open Consistency

DownClosed : {E : Set}(fpo : FinitePartialOrder E) → List E → Set
DownClosed {E} fpo es = (∀ (e e′ : E) → e ∈ es → (fpo ≤ e) e′ → e′ ∈ es)
```

2.2.3 Event structures

Event structures and their configurations can now be defined using our implementation of finite partial orders as the causality relation, plus a consistency relation.

```
record EventStructure (E : Set) : Set₁ where
  field
    fpo : FinitePartialOrder E
    Con : Consistency fpo

  config : List E → Set
config es = con-f Con es × DownClosed fpo es

open EventStructure
```

E, the type of events within an event structure, is a parameter of the record. As configurations are also specific finite sets of events, we define them in a similar manner to the consistency relation through the use of a predicate `config` that states whether a given list of events represents a valid configuration.

2.3 Simple parallel composition

As mentioned in the introduction chapter, the main appeal of using event structures as the underlying structure of a game is that event structures are able to express true concurrency. This is accomplished through the *simple parallel composition* (or just *parallel composition*) of two event structures.

2.3.1 Definition

Definition 2.3.1 (CCRW). *(Simple parallel composition of event structures). The parallel composition of two event structures E and F, written $E \parallel F$, is the event structure with:*

- Events*: The disjoint union of the events in E and F, or $\{0\} \times E \cup \{1\} \times F$ where 0 and 1 are tags.*
- Causality *($\leq$): Events originally belonging to E are subject to the causality relation of E and unaffected by the causality relation of F. The reverse holds for events originally belonging to F. More formally,*

$$(i,c) \leq_{E\parallel F} (j,c') \text{ when } i = j = 0 \text{ and } c \leq_E c', \text{ or } i = j = 1 \text{ and } c \leq_F c'$$

- Consistency*: A set of events in $E \parallel F$ is consistent if those events are consistent in either E or F. There is no conflict between the events of E and F.*

$$X \in Con_{E\parallel F} \text{ iff } \{a|(0,a) \in X\} \in Con_E \text{ and } \{b|(1,b) \in X\} \in Con_F$$

- Configurations*: There is a canonical order-isomorphism between the configurations of $E \parallel F$, and the product of the configurations of E and F. Namely,*

$$\mathcal{C}(E \parallel F) \cong \mathcal{C}(E) \times \mathcal{C}(F)$$

The above amounts to a very verbose way of stating that the parallel composition of two event structures entails juxtaposing them side-by-side without conflict or causality.

2.3.2 Composing finite partial orders

To implement parallel composition in Agda, a function $f : E \to F \to E \parallel F$ where E, F, and $E \parallel F$ are event structures must be defined in terms of the event structure record we already have. This requires defining parallel composition for finite partial orders and consistency relations, as well as proving these compositions are themselves finite partial orders and consistency relations.

Agda's built-in sum type allows us to express the disjoint union of two events, but our use of lists as representations of finite sets require some additional definitions to fully formalise the disjoint union of two sets. Therefore, we adopt the following implementation where the function `≤-f : E → List E` is instantiated with the type parameter `(E ⊎ F)` and defined by `map` -ing list items belonging to E and F to $E \uplus F$ with Agda's built-in constructors for sum types.

```
fpo-comp : ∀ {E F : Set}
→ FinitePartialOrder E → FinitePartialOrder F → FinitePartialOrder (E ⊎ F)
fpo-comp {E}{F} a b = record
                { ≤-f       = ≤-f-def
                ; ≤-f-refl  = λ { {inj₁ x} → ∈-map⁺ inj₁ (≤-f-refl a)
                                ; {inj₂ y} → ∈-map⁺ inj₂ (≤-f-refl b)
                                }
                ; ≤-f-antisym = antisym-helper
                ; ≤-f-trans = λ {_}{_}{p} → trans-helper p
                }
```

The proofs for reflexivity, anti-symmetry, and transitivity are omitted because they mostly involved proving properties about lists that are not very conceptually relevant to the objective of this project. They can be found in the source code submitted alongside this project.

2.3.3 Composing consistency relations

The composition of two consistency relations from two different event structures is a function of type:

```
con-comp : ∀ {E F : Set}
{fpo-e : FinitePartialOrder E}{fpo-f : FinitePartialOrder F}
  → Consistency fpo-e → Consistency fpo-f
  → Consistency (fpo-comp fpo-e fpo-f)
```

A `Consistency (E ∪ F)` record can be constructed similarly to the one for composing finite partial orders as detailed in section 2.3.2, where proofs of each defining property of consistency relations must be established.

These proofs, along with the predicate determining whether a given finite set is consistent, require a number of new operations on lists of $E \cup F$ for which we implement a small library in `Filter-Operations`. The following is the predicate `Con` on lists of events of type `E ⊎ F`, where `filter-e` and `filter-s` are destructors for `List (E ∪ F)` that return `List E` and `List F` respectively.

```
con-f-def : List (E ⊎ F) → Set
con-f-def xs = con-f a (filter-e xs) × con-f b (filter-f xs)
```

The rest of the property-related proofs will be omitted again because they are long and not particularly interesting.

2.3.4 Composing event structures

Given the compositions of the causality relation and consistency relation implemented above, we can now construct the parallel composition of two event structures. As stated in section 2.3.1, this composition itself should also be an event structure whose events are the disjoint union of events in E and F, giving rise to the following implementation:

```
_||e_ : ∀ {E F : Set}
→ EventStructure E → EventStructure F → EventStructure (E ⊎ F)
_||e_ {E}{F} a b = record
    { fpo  = fpo-comp (fpo a) (fpo b)
    ; Con = con-comp (Con a) (Con b)
    }
```

2.4 Event structures with polarities (games)

Definition 2.4.1 (CCRW). *(Event structures with polarities). An* event structure with polarities *(ESP), is an event structure with a* polarity relation *that maps each event $e \in E$ of the event structure onto a polarity like so:*

$$pol_E = E \rightarrow \{+, -\}$$

This definition can be directly translated into Agda, where polarities are defined as a datatype with two constructors corresponding to + and -, and ESPs are records that take an event structure and a polarity relation as fields.

```
data Pol : Set where
  + - : Pol

flip : Pol → Pol
flip + = -
flip - = +


record ESP (E : Set) : Set₁ where
  constructor mkESP
  field
    polarity : E → Pol
    event-structure : EventStructure E

open ESP
```

A *game* is an ESP that specifies the interface at which two players interact, such that game-moves are the events $e \in E$, where e^+ events are Player-moves and e^- events are Opponent-moves.

We recall from Chapter 1 that in semantic games, players make moves in strictly alternating turns. CCRW assumes this is the case for any ESP representing a game and does not place any additional requirements on the definition of ESPs to enforce the alternation of polarities. We follow their approach and use our `ESP` record directly as games in the remaining implementation work of this project.

2.4.1 Dual

Definition 2.4.2 (CCRW)**.** *(The dual of an ESP). The* dual *of a game A, written $A^\perp$, is the ESP comprising of a copy of A's underlying event structure, and a polarity relation that reverses the polarities of the events in A.*

This is easily definable in Agda as a function that takes an `ESP` as input and returns its dual, another `ESP`.

```
Dual : ∀ {E : Set} → ESP E → ESP E
Dual A = record
  { polarity = λ x → flip (polarity A x)
  ; event-structure = event-structure A
  }
```

2.4.2 Simple parallel composition

Definition 2.4.3 (CCRW)**.** *(Simple parallel composition of ESPs). We define parallel composition on two games A and B, written as $A \parallel B$, through the existing parallel composition operation on event structures and making each event $e_{A\parallel B} \in A \parallel B$ inherit the polarity from its corresponding event $e_A \in A$ or $e_B \in B$, e.g.*

$$pol_{A\parallel B}(0,a) = pol_A(a)$$
$$pol_{A\parallel B}(1,b) = pol_B(b)$$

Referring back to the $\parallel_e$ operation defined on `EventStructure` and continuing our use of Agda's sum types in place of the labelling events with $\{0,1\}$, we implement parallel composition on games like so:

```
_||_ : ∀ {E F : Set} → ESP E → ESP F → ESP (E ⊎ F)
A || B = record
   { polarity = λ { (inj₁ e) → polarity A e
                  ; (inj₂ f) → polarity B f
                  }
   ; event-structure = event-structure A ||ₑ event-structure B
   }
```

It is worth noting that parallel composition commutes with the duality operation, i.e. $(A \parallel B)^\perp = A^\perp \parallel B^\perp$.

Chapter 3

Pre-strategies

With a definition for concurrent games and their underlying event structures at hand, we now turn to the task of mechanising *strategies* on these games.

Strategies are labellings of the possible moves made by a player throughout the duration of the game, and these labellings can be represented as another ESP. Therefore, strategies are mappings from one ESP to another that obey a set of properties to ensure they represent the interface of the game. To this end we consider *pre-strategies*, upon which additional constraints can be placed to obtain a definition for strategies in the chapter following this one.

3.1 Definition

Definition 3.1.1 (CCRW). *(Pre-strategies). A pre-strategy on a game A is a function $\sigma : S \to A$, where S is another ESP that* labels A. *Pre-strategies must satisfy:*

1. *Preservation of configurations (obeying the rules of the game):*

$$\forall x \in \mathcal{C}(A),\ \sigma x \in \mathcal{C}(A)$$

2. *Local injectivity (the play is linear):*

$$\forall s, s' \in x \in \mathcal{C}(A),\ \sigma s = \sigma s' \Rightarrow s = s'$$

3. *Preservation of polarities:*

$$\forall s \in S,\ pol_A(\sigma s) = pol_S(s)$$

We note that σ does not have to preserve the ordering of events, and that σ only needs to be *locally injective* rather than injective because inconsistent events in S can map to the same event in A.

We also define total maps between two event structures.

Definition 3.1.2 (CCRW). *(Total maps of event structures). A total map between two event structures is a function on events $f : E \to F$. Total maps between event structures* *satisfy properties (1) and (2) for pre-strategies, omitting property (3) because unlike ESPs, event structures do not have polarities.*

To mechanise these definitions in Agda, we observe that pre-strategies can be defined in terms of total maps of event structures, where these event structures are lifted to ESPs and the mapping is equipped with property (3).

Definition 3.1.2 may be directly translated into Agda like so, where `σ-map` is the mapping itself and the other two fields enforce properties (1) and (2):

```
record _↦_ {S A : Set} (ESS : EventStructure S) (ESA : EventStructure A) : Set
where
field
  σ-map : S → A
  pre-1 : ∀ x → config ESS x → config ESA (map σ-map x)
  pre-2 : ∀ {e e′ : S} {es : List S} → e ∈ es → e′ ∈ es
  → config ESS es → σ-map e ≡ σ-map e′ → e ≡ e′
```

Pre-strategies are mechanised in terms of _ ↦ _ , along with the requirement that polarities are preserved:

```
record pre-strat {Sₛ Aₛ : Set} (S : ESP Sₛ) (A : ESP Aₛ) : Set₁ where
  field
    σ : (event-structure S) ↦ (event-structure A)
    pre-3 : ∀ {e : Sₛ} → polarity S e ≡ polarity A (σ-map σ e)

open pre-strat
```

3.2 Categories of event structures and ESPs

We briefly cover some category-theoretic concepts to motivate the construction of interaction between pre-strategies in CCRW.

3.2.1 Categories

Definition 3.2.1 (Leinster [2016]). *(Categories). A category $\mathcal{A}$ consists of:*

- *A collection of objects $ob(\mathcal{A})$,*
- *For each $A,B \in ob(\mathcal{A})$, a collection $\mathcal{A}(A,B)$ of* morphisms *from A to B,*
- *For each $A,B,C \in ob(\mathcal{A})$ there is a function*

$$\begin{array}{c} \mathcal{A}(B,C) \times \mathcal{A}(A,B) \to \mathcal{A}(A,C) \\ (g,f) \mapsto g \circ f, \end{array}$$

  called composition,
- *For each $A \in ob(\mathcal{A})$, an element 1_A of $\mathcal{A}(A,A)$ called the* identity *on A,*

satisfying the axioms:

- Associativity*: For each $f \in \mathcal{A}(A,B), g \in \mathcal{A}(B,C), h \in \mathcal{A}(C,D)$ we have:

$$(h \circ g) \circ f = h \circ (g \circ f)$$

- Identity laws*: For each $f \in \mathcal{A}(A,B)$, we have $f \circ 1_A = f = 1_B \circ f$.*

The identity function on an event structure E is a total map that is stable under composition. Therefore, event structures and maps form a category $\mathcal{E}$ (CCRW). Similarly, ESPs and pre-strategies (which are really polarity-preserving maps between ESPs) form a category $\mathcal{EP}$. We do not directly mechanise $\mathcal{E}$ and $\mathcal{EP}$ as categories because we do not make use of their category-theoretic properties in this project. Pen-and-paper proofs that $\mathcal{E}$ and $\mathcal{EP}$ are indeed categories can be found in CCRW.

3.2.2 Pullbacks

Definition 3.2.2 (Leinster [2016]). *(Pullbacks). Let $\mathcal{A}$ be a category with objects and maps:*

$$\begin{tikzcd} & Y \arrow[d, "t"] \\ X \arrow[r, "s"'] & Z \end{tikzcd}$$

A pullback *of this diagram is an object $P \in \mathcal{A}$ together with maps: $p_1 : P \to X$ and $p_2 : P \to Y$ such that*

$$\begin{tikzcd} P \arrow[r, "p_2"] \arrow[d, "p_1"'] & Y \arrow[d, "t"] \\ X \arrow[r, "s"'] & Z \end{tikzcd}$$

commutes, and with the property that for any commutative square

$$\begin{tikzcd} A \arrow[r, "f_2"] \arrow[d, "f_1"'] & Y \arrow[d, "t"] \\ X \arrow[r, "s"'] & Z \end{tikzcd}$$

in A, there is a unique map $\overline{f} : A \to P$ such that

$$\begin{tikzcd} A \arrow[dr, dashed, "\overline{f}"] \arrow[drr, bend left, "f_2"] \arrow[ddr, bend right, "f_1"'] & & \\ & P \arrow[r, "p_2"] \arrow[d, "p_1"'] & Y \arrow[d, "t"] \\ & X \arrow[r, "s"'] & Z \end{tikzcd}$$

commutes.

3.3 Prelude to interactions of pre-strategies

Defining and mechanising the interaction of two pre-strategies is the most complex part of this project. We therefore dedicate this section to its pre-requisites.

3.3.1 The interaction pullback

Definition 3.3.1 (CCRW)**.** *(Interaction of pre-strategies). The interaction of two pre-strategies $\sigma : S \mapsto A$ and $\tau : T \mapsto A^{\perp}$ playing on the same game A, where τ is the* counter pre-strategy *(a pre-strategy for Opponent on A) to σ (a pre-strategy for Player on A), is built from the following map between event structures:*

$$\sigma \wedge \tau = S \wedge T \to A_e$$

Here, $S \wedge T$ is an event structure describing the causal structure of events that σ and τ agree upon, and A_e is A stripped of its polarity relation. The removal of polarities is a consequence of the fact that σ and τ have opposite expectations for the polarities of events in A. Polarities will become relevant again when the composition of pre-strategies is introduced in Section 3.5.

While we do not make use of this information in our implementation, we note that interaction corresponds to a *pullback* in $\mathcal{E}$ where $S \wedge T$ is the pullback object, i.e. the following diagram commutes:

$$\begin{tikzcd}
 & S \wedge T \arrow[dl, "\Pi_1"'] \arrow[dr, "\Pi_2"] & \\
S \arrow[dr, "\sigma"'] & & T \arrow[dl, "\tau"] \\
 & A &
\end{tikzcd}$$

In the diagram above, Π_1 and Π_2 are set-theoretic projections. A proof that this is indeed a pullback can be found in CCRW.

We note the $\wedge$ symbol is overloaded: when its arguments are pre-strategies, it refers to the pre-strategy resulting from their interaction, and when the arguments are event structures it refers the pullback object illustrated above.

Given this definition, it is imperative for us to construct the event structure $S \wedge T$. This is more complicated than expected, because we cannot simply take the events of $S \wedge T$ to be those that are *synchronised* between S and T, although we define them anyway because they will come in handy in a moment:

Definition 3.3.2 (CCRW)**.** *(Synchronised events). The synchronised events of two event structures S and T are the pairs $(s,t) \in S \times T$ such that $\sigma\, s = \tau\, t$.*

The synchronised events of S and T are not simply the events of $S \wedge T$, because maps like σ and τ are *locally* injective rather than injective. For example, consider the interaction between the following two event structures:

$$\begin{tikzcd}
 & b & & & b \\
a \arrow[rr, squiggly, no head] & & a' & & a \arrow[u] \\
 & (\sigma) & & & (\tau)
\end{tikzcd}$$

where σ contains the events b and two conflicting copies of the same event $a \in A$, with no causal relation between the three. Meanwhile, τ can only play b after playing a. We can see that there are three possible synchronised pairs of events in this interaction: (a,a), (a',a), and (a,b).

However, since σ can play a in two different ways, there are actually two possible causal histories for b: b can be played after $(a,a) \in S \times T$, or $(a',a) \in S \times T$. Events in event structures are distinguished by causal histories, so there should be *four* events in $S \wedge T$ rather than the three events identified by taking synchronised pairs. This is illustrated in the following diagram of the interaction between σ and τ:

$$\begin{tikzcd}
b & b' \\
a \arrow[u] \arrow[r, squiggly, no head] & a' \arrow[u]
\end{tikzcd}$$

It turns out that directly defining the events in $S \wedge T$ is quite difficult. Therefore, we construct $S \wedge T$ indirectly by specifying the set of configurations it should have as *prime secure bijections* between the configurations of $S \wedge T$ and the configurations of the synchronised events of S and T.

3.3.2 Prime secure bijections

Referring back to the diagrams above, we note there is a one-to-one correspondence between the desired configurations of $S \wedge T$ and the configurations of the synchronised events. Writing synchronised configurations as pairs:

$$(x,y) \in \mathcal{C}(S) \times \mathcal{C}(T) \text{ where } \sigma x = \tau y,$$

We observe that by local injectivity, σ and τ induce a bijection $\phi : x \simeq \sigma x = \tau y \simeq y$, whose graph is the set of synchronised pairs of events as described above (CCRW).

We use a set of secured bijections ϕ as the set of configurations of $S \wedge T$ with the caveat that ϕ itself is not ordered like a configuration of an event structure should be, so we must define this ordering. Our desired ordering should be inherited from the orderings in S and T, but it is not simply their transitive closure. Consider the following situation where a *deadlock* occurs:

$$\begin{tikzcd} \text{PhD position} & \text{funding} \\ \text{funding} \arrow[u] & \text{PhD position} \arrow[u] \\ (\sigma) & (\tau) \end{tikzcd}$$

A prospective PhD student who is only able to obtain a position upon being funded (σ) patiently waits for a scholarship to materialise, while a university (τ) only offers funding to students who have already secured PhD positions. There is no event the two agree upon, and so their interaction only contains the empty configuration. Given this state of affairs, the student will never obtain the PhD position and no funding will ever be distributed.

The reason why the preorder induced by S and T on the bijection

$$\{(\text{PhD position, PhD position}), (\text{funding,funding})\}$$

is not an order is that there is a loop. These loops can be eliminated by introducing the notion of *secured bijections*.

Definition 3.3.3 (CCRW)**.** *(Secure bijections). A bijection $\phi : q \simeq q'$ between two finite orders $(q, \leq_q)$ and $(q', \leq_{q'})$ is* secured *if the reflexive and transitive closure of the following relation on the graph of ϕ is an order:*

$$(a,b) \lhd (a',b') \text{ if } a <_q a' \text{ or } b <_{q'} b'$$

This order is reflexive and transitive by definition. It is also anti-symmetric, which eliminates the possibility of deadlock loops.

To construct a mechanised definition for secure bijections, we begin with the observation that a bijection is a set of pairs (s,t) where each s and t are in one-to-one correspondence. Since we are only concerned with bijections between finite orders, it follows that we should mechanise bijections through our existing implementation of finite sets — that is, through a list of pairs. We then enforce that the bijection is secure through additional properties it must satisfy.

The arguments ϕ, q and q′ of the following module correspond to ϕ, q and q' in Definition 3.3.4.

```
module _{Q Q′ : Set}
  {Qₑ : EventStructure Q}{Qₑ′ : EventStructure Q′}
  (ϕ : List (Q × Q′))(q : List Q) (q′ : List Q′) where
  open FinitePartialOrder
  data _◁*_ : (Q × Q′) → (Q × Q′) → Set where
    reflexive : ∀{p} → p ◁* p
    transitive : ∀{p q r} → p ◁* q → q ◁* r → p ◁* r
    step₁ : ∀{a}{b}{a′}{b′}
      → (a , b) ∈ ϕ
      → (a′ , b′) ∈ ϕ
      → _≤_ (fpo Qₑ) a a′
      → (a , b) ◁* (a′ , b′)
    step₂ : ∀{a}{b}{a′}{b′}
      → (a , b) ∈ ϕ
      → (a′ , b′) ∈ ϕ
      → _≤_ (fpo Qₑ′) b b′
      → (a , b) ◁* (a′ , b′)
```

Each constructor for the $\lhd*$ datatype corresponds to a property listed in the definition: `step1` and `step2` are a direct translation of the relation described in Definition 3.3.4, while `reflexive` and `transitive` ensure the datatype is a reflexive transitive closure.

We also construct a record $\simeq$ that ensures the $\lhd*$ relation is indeed a (secure) bijection, and that it only takes configurations of event structures Q_e and Q'_e as arguments:

```
  record _: _≃_ : Set where
    field
      config-left : config Qₑ q
      config-right : config Qₑ′ q′
      coverage-left : filter-e-× ϕ ≡ₛ q
      coverage-right : filter-f-× ϕ ≡ₛ q′
      bijection-left : ∀{a}{b c} → (a , b) ∈ ϕ → (a , c) ∈ ϕ → b ≡ c
      bijection-right : ∀{a b}{c} → (a , c) ∈ ϕ → (b , c) ∈ ϕ → a ≡ b
      secured : ∀{a a′}{b b′} → (a , b) ◁* (a′ , b′)
                   → (a′ , b′) ◁* (a , b) → (a , b) ≡ (a′ , b′)
```

The definitions for $\lhd*$ and $\simeq$ are grouped in the same module to ensure that both sets of requirements apply to the same two event structures Q_e and Q'_e. The helper functions `filter-e-×` and `filter-f-×` are deconstruct `List E × F` into `List E` and `List F` respectively.

We now use secured bijections between σ and τ to describe the set $\mathcal{B}^{sec}_{\sigma\tau}$, the configurations of $S \wedge T$ (CCRW):

$$\mathcal{B}^{sec}_{\sigma\tau} = \{\phi \mid \phi : x \overset{\sigma}{\simeq} \sigma x = \tau y \overset{\tau}{\simeq} y \text{ is secured, with } x \in \mathcal{C}(S),\ y \in \mathcal{C}(y)\}$$

This definition translates nicely into Agda, where we divide ϕ into three components: the bijections $(x, \sigma x)$, $(\tau y, y)$, and the equality $\sigma x = \tau y$. We then combine these components as a product type (i.e. conjunction) to construct a predicate that determines if a given set of pairs is the $\mathcal{B}^{sec}_{\sigma\tau}$ set for some σ and τ.

```
  Bsec : ∀ {Sₛ Tₛ Aₛ : Set}
           {S : EventStructure Sₛ} {T : EventStructure Tₛ} {A : EventStructure Aₛ}
  → (σ : S ↦ A) → (τ : T ↦ A) → List (S_s × T_s) → Set
Bsec {S = S}{T}{A} σ τ ϕ =
  _: _≃_ (map (λ (p , _) → ( p , σ-map σ p )) ϕ)
           (filter-e-× ϕ) (map (σ-map σ) (filter-e-× ϕ))
  × _: _≃_ (map (λ (_ , q) → ( q , σ-map τ q )) ϕ)
           (filter-f-× ϕ) (map (σ-map τ) (filter-f-× ϕ))
  × All (λ ( x , y) → σ-map σ x ≡ σ-map τ y) ϕ
```

The order $(\mathcal{B}^{sec}_{\sigma\tau}, \subseteq)$ gives the order of configurations for the event structure $S \wedge T$. Through this, we can identify the set of events belonging to $S \wedge T$ as those that are *prime secure bijections*.

Definition 3.3.4 (CCRW)**.** *(Prime secure bijections). Prime secure bijections are secure bijections with a* top event*, i.e. a greatest synchronised pair* (s,t).

The absence of a top event in the causal history of a synchronised pair indicates a loop or deadlock. By taking the subset of $(\mathcal{B}^{sec}_{\sigma\tau}, \subseteq)$ that are prime secure bijections to be the configurations of $S \wedge T$, we eliminate the problem previously illustrated with the example involving the PhD applicant's unfortunate funding situation.

The mechanisation of prime secure bijections is by far the most challenging part of our implementation work, and the only place we employ a postulate. We dedicate the following section towards it.

3.3.3 Postulate: Canonically-ordered lists

Three postulates are required as an extension to our implementation of finite sets through lists. In the previous parts of our implementation, we have gotten away with defining finite set equality via list inclusion like the following:

```
_⊆_ : ∀ {A : Set} → List A → List A → Set
a ⊆ b = All (_∈ b) a


_≡s_ : ∀ {A : Set} → List A → List A → Set
a ≡s b = (a ⊆ b) × (b ⊆ a)
```

Here, $\equiv_s$ is an equivalence relation on lists that ignores duplicate items and the order items appear within the list, which provides the set equality operation we need.

The definitions above are inadequate for mechanising prime secure bijections because it is different from propositional equivalence. Propositional equality, also called the identity type, is the proposition that captures intensional equality between two terms in type theory. In Agda, it is denoted by the $\equiv$ symbol.

The problem with our definition of set equality arises later in Definition 3.4.1, where we will define the event structure $S \wedge T$'s events to be *the elements of* $\mathcal{B}^{sec}_{\sigma\tau}$ *that have a top event*. We recall that $\mathcal{B}^{sec}_{\sigma\tau}$ is a set of secured bijections, and that we implement secured bijections as lists of products, e.g. `List (S × T)`. Meanwhile, in the `EventStructure` record we encode the anti-symmetry property of finite partial orders using propositional equality. Propositional equality on `List (S × T)` is list equality rather than our custom-defined set equality. We have no real way of telling Agda that we want it to use $\equiv_s$ in this specific instance but stick to propositional equality in others, without a major overhaul of our definition for event structures that is infeasible given the time constraint on this project.

Therefore, we circumvent the problem described above by employing three postulated theorems stating that the lists we are using as finite sets have a *canonical order*.

```
postulate
  canonical : {A : Set} → List A → List A
  canonical-s : ∀{A}{ls : List A} → ls ≡ₛ canonical ls
  canonical-t : ∀{A}{ls₁ ls₂ : List A}
    → ls₁ ≡ₛ ls₂ → canonical ls₁ ≡ canonical ls₂
```

Here,

- `canonical` takes a list and returns its canonical form,
- `canonical-s` states that a given list is equal to its canonical form by our custom definition of set equality, and
- `canonical-t` states that given two lists equivalent by set equality, their canonical forms are equal by propositional equality.

Combined, the three postulated theorems mean all lists in canonical form represent finite sets that can be equated through propositional equality. Since some `Sets` are not ordered and therefore cannot have a canonical ordering, these theorems must be postulated rather than proved. Overall, we do not believe these postulates are a big problem for our project because we know they can be eliminated given more time and effort, and their necessity reflects Agda's lack of an existing library for finite sets.

3.3.4 Back to prime secure bijections

With the canonicity postulates out of the way, we return to our mechanisation of prime secure bijections by constructing a record `S∧T-config` to wrap our previously-defined predicate `Bsec` together with the `canonical` postulate. This ensures propositional equality on `S∧T-config` is set equality, not list equality.

```
record S∧T-config {Sₛ Tₛ Aₛ : Set} {S : EventStructure Sₛ}
  {T : EventStructure Tₛ} {A : EventStructure Aₛ}
  (σ : S ↦ A) (τ : T ↦ A) : Set where
  constructor mk-config
  field
    Φ : List (Sₛ × Tₛ)
    .bsec : Bsec σ τ Φ
    is-c : Φ ≡ canonical Φ

open S∧T-config
```

Now, we must specify those members of `S∧T-config` that have a top event. We begin by defining the notion of a top element as a predicate on pairs of events in `Bsec` with respect to two arbitrary event structures.

```
IsTop : {Sₛ Tₛ Aₛ : Set} {S : EventStructure Sₛ}
        {T : EventStructure Tₛ} {A : EventStructure Aₛ}
        {σ : S ↦ A} {τ : T ↦ A}
        (event : S∧T-config σ τ) → (Sₛ × Tₛ) → Set
IsTop {Sₛ}{Tₛ}{Aₛ}{S}{T}{A} event (s , t) = All (λ { (s′ , t′)
  → ((fpo S) ≤ s′) s × ((fpo T) ≤ t′) t}) (Φ event)
```

The predicate only `IsTop` holds for pairs of events $(S_s \times T_s)$ where both components are the greatest element in their respective causality orderings. Now, we encode prime secure bijections as the elements of `pre-S∧T-config` satisfying `IsTop`:

```
record PrimeSecBij {Sₛ Tₛ Aₛ : Set} {S : EventStructure Sₛ}
                   {T : EventStructure Tₛ} {A : EventStructure Aₛ}
                   (σ : S ↦ A) (τ : T ↦ A) : Set where
  constructor mk-psb
  field
    event : S∧T-config σ τ
    top : Sₛ × Tₛ
    .prf : IsTop event top

open PrimeSecBij
```

3.4 The interaction of pre-strategies

As promised, we are now ready to construct $S \wedge T$. We first re-iterate the interaction between two maps of event structures $\sigma : S \to A$ and $\tau : T \to A$ given in Definition 3.1.1:

$$\sigma \wedge \tau = S \wedge T \to A_e$$

Definition 3.4.1 (CCRW)**.** *(Interaction of pre-strategies). We construct the event structure $S \wedge T$ as the following:*

- Events: *the elements of $\mathcal{B}^{sec}_{\sigma\tau}$ that have a top event,*
- Causality: *graph inclusion,*
- Consistency: *A finite set of secure bijections is consistent when their union is also a secure bijection in $\mathcal{B}^{sec}_{\sigma\tau}$.*

This is where things get very, very hairy for our implementation because constructing an `EventStructure` record entails proving that its associated causality and consistency relations satisfy the properties encoded in `FinitePartialOrder` and `Consistency`. This has not been a problem for the previous sections of the project because the proofs required were of a manageable length. In our definition for the `S∧T` event structure, these proofs explode in complexity and the number of supporting lemmas they require.

Therefore, from this point onwards we adopt the streamlined approach of only filling in those parts of the record directly corresponding to the formal definitions provided in CCRW. We leave proofs of the associated properties as future work because it is infeasible to complete them within the given timeframe, and we do not believe their absence compromises the consistency of the project.

With the above said, we define a function `S∧T` that takes two suitable event structure mappings as arguments and returns the event structure $S \wedge T$:

```
S∧T : {Sₛ Tₛ Aₛ : Set} {S : EventStructure Sₛ} {T : EventStructure Tₛ}
      {A : EventStructure Aₛ} (σ : S ↦ A) (τ : T ↦ A)
      → EventStructure (PrimeSecBij σ τ)
S∧T {Sₛ}{Tₛ}{Aₛ}{S}{T}{A} σ τ = record
  { fpo = record
          { ≤-f         = λ x → fpo-≤-f x
          ; ≤-f-refl    = refl-help
          ; ≤-f-antisym = antisym-help
          ; ≤-f-trans   = λ {m}{n}{p} q r → trans-help {p}{z = m} r q
          }
  ; Con = record
          { con-f       = λ x → Bsec σ τ
                            (concat (map (λ y → Φ (event y)) x))
          ; con-1       =  {!!}
          ; con-2       =  {!!}
          ; con-3       =  {!!}
          ; con-empty   =  {!!}
          }
          }
```

While the construction above is incomplete, each feature of $S \wedge T$ given in Definition 3.4.1 is addressed:

- *Events*: the events of `S∧T` have type `TopSet σ τ`, which is our encoding of those elements of $\mathcal{B}^{sec}_{\sigma\tau}$ with a top event.
- *Consistency*: The `con-f` field contains the consistency relation of `S∧T`. Since `Bsec` ensures the lists in `TopSet` are `canonical`, list concatenation on elements of `TopSet` is equivalent to set union. Therefore, `con-f` can be read as a predicate stating whether the set union of a finite number of values with type `TopSet σ τ` are in `Bsec`.
- *Causality*: The causality relation `≤-f` is outsourced to a helper function `fpo-≤-f x`, defined as the following:

```
fpo-≤-f : TopSet σ τ → List (TopSet σ τ)
```

```
fpo-≤-f x = mapMaybe toTopSet (mapMaybe toConfig'
(map canonicalise (powerset (Φ (event x)))))
```

To understand what this function does, we re-iterate that `TopSet` is a wrapper for a list of pairs, and so `fpo-≤-f` returns a list of lists of pairs. We are interested in defining graph inclusion on the elements of this list of lists, which we note is equivalent to set inclusion in our implementation. Therefore, we implement graph inclusion as a function that maps a given value of type `TopSet σ τ` to a list comprising of members of its `powerset` (or, sub-lists) that have a top event.

We omit explanations of the other helper functions used in this definition because their names should be fairly self-explanatory.

3.5 Composition of pre-strategies

Given we have a working definition for the interaction of two pre-strategies (Definition 4.2.1 along with the construction of $S \wedge T$), we are now ready to define what it means for pre-strategies to be composed. This composition operation is vital to the primary goal of this project, that being the mechanisation of composable strategies.

It matters that strategies are composable because we ultimately want *games* to be composable, such that they can be useful in the context of denotational semantics. For this purpose, the interaction described in Definition 4.2.1 must be expanded from pre-strategies playing on the same game A to pre-strategies playing *from* a game A to another game B.

More formally, what we currently have are interactions between two pre-strategies $\sigma_A : S \to A$ and $\tau_a : T \to A$, where S, T, and A are ESPs. We would now like to specify this definition to the interaction between pre-strategies $\sigma : S \to A^{\perp} \parallel B$ and $\tau : T \to B^{\perp} \parallel C$ on the composite game $A^{\perp} \parallel C$. This is accomplished through two steps: defining the interaction of our new versions of σ and τ, then performing a *hiding* operation on the resulting pre-strategy.

3.5.1 Interaction of pre-strategies on a composite game

Definition 3.5.1 (CCRW). *(Interaction). We temporarily forget about the polarity relations on σ and τ since they can be easily recovered. Then, σ and τ are respectively mappings between event structures $S \to A \parallel B$ and $T \to B \parallel C$. Since pre-strategies can only interact if they play on the same game (e.g. map to ESPs with the same underlying event structure), we pad σ and τ with identity maps so that they both map to $A \parallel B \parallel C$:*

$$\sigma \parallel id_C : S \parallel C \to A \parallel B \parallel C$$

$$id_A \parallel \tau : A \parallel T \to A \parallel B \parallel C$$

The interaction of these padded pre-strategies is:

$$(\sigma \parallel id_C) \wedge (id_A \parallel \tau) : (S \parallel C) \wedge (A \parallel T) \to A \parallel B \parallel C$$

We introduce an interaction operator ⊛ to simplify the notation of the above to:

$$\tau \circledast \sigma : T \circledast S \to A \parallel B \parallel C$$

The positions of σ and τ are reversed when the ⊛ operator is used, reflecting the standard notation for composition. CCRW overloads the ⊛ notation in a similar way to the ∧ notation introduced in the previous section: it refers to the event structure mapping resulting from the interaction of σ and τ when its arguments are pre-strategies, and otherwise refers to the relevant event structure.

We mechanise ⊛ as a function taking maps of event structures τ and σ as arguments, to return the mapping τ ⊛ σ given in Definition 3.5.1:

```
_⊛_ : ∀ {Sₛ Tₛ Aₛ Bₛ Cₛ : Set} {S : EventStructure Sₛ}
  {T : EventStructure Tₛ} {A : EventStructure Aₛ}
  {B : EventStructure Bₛ} {C : EventStructure Cₛ}
  → (σ : (S ||ₑ C) ↦ ((A ||ₑ B) ||ₑ C))
  → (τ : (A ||ₑ T) ↦ ((A ||ₑ B) ||ₑ C))
  → (S∧T σ τ ↦ ((A ||ₑ B) ||ₑ C) )
τ ⊛ σ = record
  { σ-map = λ { (mk-topset event₁ (s , t) prf) → σ-map σ t}
  ; pre-1 = {!!}
  ; pre-2 = {!!}
  }
```

The σ-`map` field in the resulting _↦_ record contains a function of type `S∧T σ τ → ((A` $\|_e$ `B)` $\|_e$ `C)`, which we simply define to be identical as the σ-`map` field in σ. It does not matter whether we use the mapping contained in σ or τ here, because the events of `S∧T` are products where both members of the pair are mapped onto the same value by σ and τ.

Then, `pre-1` enforces the preservation of configurations and `pre-2` ensures σ-`map` is a bijection. The proofs that our encoding of τ ⊛ σ satisfy `pre-1` and `pre-2` are left incomplete because they are dependent on the incomplete proofs in `S∧T`, without which we cannot proceed.

3.5.2 Hiding

Definition 3.5.1 gives the interaction of two event structure mappings on $A \parallel B \parallel C$, which isn't yet the pre-strategy on $A^{\perp} \parallel C$ we want. We now introduce a *hiding* operation that projects τ ⊛ σ to a map to $A \parallel C$, and reinstate polarities to obtain the desired pre-strategy.

Definition 3.5.2 (CCRW)**.** *(Hiding). Events $p \in T \circledast S$ mapped to events in A or C are* visible*, and* invisible *otherwise. We write the* hiding *of invisible events in $T \circledast S$ as $(T \circledast S) \downarrow V$, where V is the set of visible events in $T \circledast S$.*

Our mechanisation and proofs for the hiding operation are complete because they are independent from the mechanisation of $S \wedge T$ we left unfinished. We first introduce a datatype representing the ↓ operation that filters for all inhabitants of a `Set` satisfying a given predicate `f`.

```
data _↓_ (A : Set) (f : A → Bool) : Set where
  filtered : (x : A) → (T (f x)) → A ↓ f
```

Then, we define a function `inj1or3` that determines whether a given input of type $A \cup B \cup C$ is inhabited by a value of type A or C, such that we can combine it with the ↓ datatype to discard those of type B.

```
inj1or3 : {A B C : Set} → (A ⊎ B) ⊎ C → Bool
inj1or3 (inj₁ (inj₁ x)) = true
inj1or3 (inj₁ (inj₂ y)) = false
inj1or3 (inj₂ y) = true
```

We note that both `inj1or3` and the predicate `f` in the ↓ datatype are implemented as functions with type `A → Bool` rather than `A → Set`. This is because we only ever use ↓ on `inj1or3`, a function with only has two possible outputs, so we don't need the additional structure that comes with the `Set`-based approach.

Equipped with the above, we can now define a function that performs the hiding operation on σ, a mapping between event structures. We omit the body of the function from this report because it is very long.

```
hide-e : ∀ {Xₛ Aₛ Bₛ Cₛ : Set}
  {X : EventStructure Xₛ} {A : EventStructure Aₛ}
  {B : EventStructure Bₛ} {C : EventStructure Cₛ}
  → (σ : X ↦ ((A ||ₑ B) ||ₑ C)) → (EX : EventStructure Xₛ)
  → EventStructure (Xₛ ↓ λ x → inj1or3 (σ-map σ x))
```

The event structure resulting from applying `hide-e` to σ : `X ↦ ((A` $\|_e$ `B)` $\|_e$ `C))` is `X` where all events mapping to `B` are removed. Through this, we can now write

$$\texttt{(hide-e } \sigma \texttt{ X)} \mapsto \texttt{(A } \|_e \texttt{ C)}$$

as the projection of σ to its subset that only includes the mappings to `(A` $\|_e$ `C)`.

3.5.3 Composition

Equipped with the interaction and hiding operations on pre-strategies, we can now define their composition.

Definition 3.5.3 (CCRW)**.** *(Composition of pre-strategies). The composition of two pre-strategies $\sigma : S \to A^{\perp} \parallel B$ and $\tau : T \to B^{\perp} \parallel C$ is:*

$$\tau \odot \sigma : T \odot S \to A^{\perp} \parallel C$$

where

$$T \odot S = (T \circledast S) \downarrow V$$

The polarity relation on $T \odot S$ is directly inherited from the polarity relation of $A^{\perp} \parallel C$.

We remark the $\odot$ operator is once again overloaded in the same manner as $\circledast$ and $\wedge$.

Our mechanisation of the composition of pre-strategies is partial because it is dependent on the incomplete implementations of $S \wedge T$ and $\circledast$. Therefore, we continue the previous approach of only mechanising definitions and forgoing proofs for the properties they satisfy.

We begin by constructing the event structure mapping $T \odot S \rightarrow A \parallel C$, where we apply our `hide-e` operation on $T \circledast S$ to obtain the subset of $T \circledast S$ that maps to $A \parallel C$.

```
_⊙-map_ : ∀ {S_s T_s A_s B_s C_s : Set} {S : EventStructure S_s}
  {T : EventStructure T_s} {A : EventStructure A_s}
  {B : EventStructure B_s} {C : EventStructure C_s}
  → (σ : (S ||_e C) ↦ ((A ||_e B) ||_e C))
  → (τ : (A ||_e T) ↦ ((A ||_e B) ||_e C))
  → hide-e (_⊛_ σ τ) (S∧T σ τ) ↦ (A ||_e C)
```

Since function names cannot be overloaded in Agda, we name this function $\odot_e$ rather than just $\odot$. We omit the body of this function because the implementation is long and incomplete.

We would now like to construct a pre-strategy from this mapping by reinstating polarities on $T \odot S$, which we recall should be identical to the polarity relation of $A^{\perp} \parallel C$. Therefore, we define a function that constructs an ESP from the domain of an event structure mapping, whose polarity relation is taken from

```
Polarise : ∀ {A_s B_s : Set} → (A_e : EventStructure A_s) → (B : ESP B_s)
  → (m : A_e ↦ (event-structure B)) → ESP A_s
Polarise A_e B m = (mkESP (λ x → polarity B (σ-map m x)) A_e)
```

Then, we use `Polarise` to define a second function `polarise` with a lower-case `p`, which directly constructs a pre-strategy from a given mapping of event structures, provided we already have access to the polarities of the ESP containing the event structure being mapped onto.

```
polarise : ∀ {A_s B_s : Set} {A_e : EventStructure A_s} {B : ESP B_s}
  → (m : A_e ↦ (event-structure B)) → pre-strat (Polarise A_e B m) B
polarise m = record
  { σ = m
  ; pre-3 = refl
  }
```

Now, we are ready to define $\tau \odot \sigma$. To make the code more readable, we decompose the definition into $\odot$ itself, and a separate construction for the ESP that is the polarisation of $T \odot S$. The construction of an ESP with $T \odot S$ as the underlying event structure is accomplished through the following function:

```
T⊙S : ∀ {S_s T_s A_s B_s C_s D_s : Set} {S : ESP S_s} {T : ESP T_s}
  {A : ESP A_s} {B : ESP B_s} {C : ESP C_s}
  → (a : pre-strat T ((Dual B) || C))
  → (b : pre-strat S ((Dual A) || B))
  → ESP {!TopSet (σ a) (σ b)!}
T⊙S {S = S} {T} {A}{B}{C} τ σ =
  Polarise (hide-e (_⊛_ (mapping-σ σ τ)
            (τ-assoc (mapping-τ σ τ)))
            (S∧T (mapping-σ σ τ) (τ-assoc (mapping-τ σ τ))))
            ((Dual A) || C)
             {!((mapping-σ σ τ) ⊙-map (τ-assoc (mapping-τ σ τ)))!}
```

Here, `mapping-τ` and `mapping-σ` are τ and σ padded with identity maps, as described in Definition 3.5.1. We omit the details of their Agda mechanisations from this report in the interest of space.

The other helper function used here is `τ-assoc`, which re-associates

$$\texttt{(A} \parallel_e \texttt{T)} \mapsto \texttt{(A} \parallel_e \texttt{(B} \parallel_e \texttt{C))}$$
to
$$\texttt{(A} \parallel_e \texttt{T)} \mapsto \texttt{((A} \parallel_e \texttt{B)} \parallel_e \texttt{C).}$$

The associativity of the parallel composition of event structures is one of the many lemmas left implicit in CCRW that need to be explicitly proven in Agda. Now, we can use this ESP to mechanise the ⊙ operation:

```
_⊙_ : ∀ {Ss Ts As Bs Cs : Set} {S : ESP Ss} {T : ESP Ts}
  {A : ESP As} {B : ESP Bs} {C : ESP Cs}
  → (τ : pre-strat T ((Dual B) || C))
  → (σ : pre-strat S ((Dual A) || B))
  → pre-strat (T⊙S τ σ) ((Dual A) || C)
_⊙_ σ τ = polarise
            {!((mapping-σ σ τ) ⊙-map (τ-assoc (mapping-τ σ τ)))!}
```

The body of the _⊙_ function composes the previously-defined _⊙`-map`_ with `polarise` to yield the desired pre-strategy τ ⊙ σ.

One final thing we note about the two mechanisations above is that all calls to `((mapping-σ σ τ) ⊙-map (τ-assoc (mapping-τ σ τ)))` appear between brackets of the form `{! !}`. This is because we are leaving them as open holes in our proofs so that our code compiles without actually typechecking their contents. We adopt this approach because these calls appear to trigger a pathological case in Agda's unification engine as it solves constraints to check that our code is well-typed: either it does not terminate, or it performs too poorly for us to know if it terminates within a reasonable timeframe. While we were not able to circumvent the problem, we typechecked the hole submissions in part and are confident they are the correct definitions. In future work, we hope to derive a minimal working example and report this case to the Agda development team.

Chapter 4

Strategies

In this chapter we describe our mechanisation of the copycat pre-strategy, isomorphisms between pre-strategies, and strategies themselves.

To motivate the copycat pre-strategy, we once again recall the underlying mathematical objects of denotational semantics are compositional, so compositionality is the guiding principle to the way in which strategies are defined in this chapter.

Now that compositionality has already been obtained for pre-strategies, we require an identity on them. This identity is the *copycat (pre)-strategy*, which acts as an *asynchronous forwarder* between pre-strategies. Then, strategies can be defined as those pre-strategies that are invariant under composition with copycat.

4.1 The copycat pre-strategy

The copycat pre-strategy plays on the composite game $A^{\perp} \parallel A$, with the role of Player in one subgame and Opponent in the other. For every Opponent-move made in $A^{\perp}$, copycat plays the exact same move on A, waits for the next Player-move made in A, then plays this Player-move on $A^{\perp}$. In doing so, the copycat is really playing the Opponent in $A^{\perp}$ against the Player in A, such that the addition of the copycat has no effect on the moves being played in A except they are now being split across two games. This should motivate the use of the copycat pre-strategy as the identity on pre-strategies, for which we now provide a formal definition.

Definition 4.1.1 (CCRW)**.** *(The copycat event structure). The copycat event structure on a game A, written* $\mathbf{C}_e^A$*, has:*

- *Events: the events of* $A^{\perp} \parallel A$*,*
- *Causality: the transitive closure of:*

  $$\leq_{A^{\perp}\parallel A} \cup \{(\overline{c}, c) \mid c \in A^{\perp} \parallel A \text{ and } pol_{A^{\perp}\parallel A}(c) = +\}$$

  where $\overline{c}$ *denotes the copy of an event* $c \in A^{\perp}$ *with the opposite polarity, in the component of the composition that c itself does not belong to.*

- *Consistency: A set of events $X \in \mathbf{C^A_e}$ is consistent iff its down-closure, i.e.*

$$\{a \in \mathbf{C^A_e} \mid \exists b \in X.\, a \leq_{\mathbf{C^A_e}} b\}$$

  is consistent in $A^\perp \parallel A$.

Definition 4.1.2 (CCRW)**.** *(The copycat pre-strategy). By adding the polarity relation from $A^\perp \parallel A$ to $\mathbf{C^A_e}$, the copycat event structure on a game A, we obtain an ESP C^A. Then, the following identity map is the* copy-cat pre-strategy *on A:*

$$\mathbf{c}^A : \mathbf{C}^A \to A^\perp \parallel A$$

Compared to the difficulty of mechanising the interaction and composition of pre-strategies in Agda, the copycat pre-strategy lends itself well to mechanisation using the machinery we have developed for event structures, ESPs, and pre-strategies over the past two chapters.

In the following code snippet, the copycat ESP $\mathbf{C}^A$ takes the definition for the copycat event structure $\mathbf{C}^A_e$ as a field because there is no need to define $\mathbf{C}^A_e$ as an independent record.

```
copycat-esp : ∀ {As : Set} → (A : ESP As) → ESP (As ⊎ As)
copycat-esp {As} game = record
  { polarity = polarity (Dual game || game)
  ; event-structure = record
   { fpo = record
     { ≤-f        = ≤-f-def
     ; ≤-f-refl   = λ{ {x} → ≤-refl-help x }
     ; ≤-f-antisym = λ x x₁ → ≤-antisym-help _ _ x x₁
     ; ≤-f-trans = λ { {p = p} x x₁ → ≤-trans-help {p = p} x x₁}
     }
   ; Con = record
      { con-f = con-f (Con ((event-structure game ||ₑ event-structure game)))
      ; con-1 = λ { {e} → con-1
                ((Con
                ((event-structure game ||ₑ event-structure game)))) {e}}
      ; con-2 = con-2 (Con ((event-structure game ||ₑ event-structure game)))
      ; con-3 = λ es x x₁ x₂ →  {!!}
      ; con-empty = con-empty
         (Con ((event-structure game ||ₑ event-structure game)))
      }
   }
```

The events of $\mathbf{C}^A$ and $\mathbf{C}^A_e$ are two copies of the events of A, expressed as the sum type $A \cup A$ in Agda. As advertised in Definition 5.1.2, $\mathbf{C}^A$ inherits its polarity and consistency relations from $A^\perp \parallel A$, and the causality relation $\leq -$f is the only part of the record that requires a new definition. This is reflected by the fact that all fields of the consistency relation Con that do not make use of the $\leq -f$ relation are identical to their definitions in $A^\perp \parallel A$.

The causality relation $\leq$ `-f` is outsourced to a helper function $\leq$ `-f-def` because of its complexity. We recall from Definition 5.1.1 that $\leq$ `-f-def` should be the transitive closure of the union of the causality relation of $A^{\perp} \parallel A$ and all pairs of events $(\overline{c}, c)$ where c has positive polarity in $A^{\perp} \parallel A$. Making use of our implementation of finite sets as lists again, $\leq$ `-f-def` may be defined as:

```
≤-f-def : As ⊎ As → List (As ⊎ As)
≤-f-def (inj₁ x) with (polarity (Dual game ∥ game)) (inj₁ x)
... — + = (≤-f (fpo (event-structure (Dual game ∥ game))) (inj₁ x))
          ++ (≤-f (fpo (event-structure (Dual game ∥ game))) (inj₂ x))
... — - = (≤-f (fpo (event-structure (Dual game ∥ game)))) (inj₁ x)
≤-f-def (inj₂ y) with (polarity (Dual game ∥ game)) (inj₂ y)
... — + = ((≤-f (fpo (event-structure (Dual game ∥ game)))) (inj₂ y))
          ++ (≤-f (fpo (event-structure (Dual game ∥ game))) (inj₁ y))
... — - = (≤-f (fpo (event-structure (Dual game ∥ game)))) (inj₂ y)
```

Here we use Agda's pattern-matching in two different ways: Firstly, we distinguish between events with positive and negative polarities in $A^{\perp} \parallel A$ through the use of a `with`-abstraction. For a given event $c \in A^{\perp} \parallel A$, the finite set of events $X = \{x \mid x \leq c\}$ is unchanged if c has negative polarity. If c has positive polarity, we must append the causal dependencies of $\overline{c}$ to X as well as c's own dependencies from $A^{\perp} \parallel A$. To implement this, we identify $\overline{c}$ (the copy of c in the component of $A^{\perp} \parallel A$ that c does not appear in) by case-matching on whether c has type $inj_1\, x$ or $inj_2\, x$ (the two constructors for sum types), because $\overline{c}$ must be x wrapped by the opposite constructor.

Secondly, we take the union of $\overline{c}$ and c's causal dependencies in $A^{\perp} \parallel A$ as the causal dependency for c in $\mathbf{C}_e^A$. In the code snippet above this is accomplished through list concatenation, owing to our implementation of finite sets as lists.

Agda proofs that $\mathbf{C}_e^A$ obeys our existing interface for event structures are omitted from this report because they are more than 200 lines long and mostly concern proofs of list properties.

Now, we define an event structure mapping from $\mathbf{C}^A$ to the game $A^{\perp} \parallel A$.

```
copycat-map : ∀ {As : Set} → (A : ESP As)
  → event-structure (copycat-esp A) ↦ event-structure (Dual A ∥ A)
copycat-map {As} a = record
  { σ-map = λ x → x
  ; pre-1 = λ { x ((con-f-a₁ , con-f-a₂) , f)
            → (pre-1-help-e {x} con-f-a₁ , pre-1-help-f {x} con-f-a₂),
            λ e e′ e∈xs e′∈exs → map-∈-eq {e}{e′}
            (f e e′ (∈-map-eq {e′}{e} e∈xs)
                    (pre-1-help {e}{e′} e′∈exs)) }
  ; pre-2 = λ { _ _ _ refl → refl}
  }
```

Here, the field σ`-map` is the identity function because $\mathbf{C}^A$ and `Dual A` $\parallel$ `A` have the exact same set of events. We then use this mapping to construct the copycat pre-strategy itself:

```
copycat-strat : ∀ {As : Set} → (A : ESP As)
  → pre-strat (copycat-esp A) (Dual A ∥ A)
copycat-strat a = record
  { σ = copycat-map a
  ; pre-3 = refl
  }
```

The `pre-3` field, corresponding to the preservation of polarities, is discharged with the `refl` constructor for propositional equality as we have already instructed Agda that `copycat-esp` has the exact same polarity relation as `Dual A ∥ A`.

4.2 Isomorphism of pre-strategies

Before arriving at our goal of defining concurrent strategies, we define and mechanise isomorphisms between pre-strategies.

Definition 4.2.1 (CCRW)**.** *(Isomorphism of pre-strategies). Let $\sigma : S \to A$ and $\tau : T \to A$ be two pre-strategies playing on the same game A. σ and τ are isomorphic if there is an isomorphism ϕ between the event structures S and T, i.e. $\phi : S \cong T$, commuting with the actions σ and τ on the game like so:*

$$\begin{tikzcd}
S \arrow[rr, leftrightarrow, bend left, "\phi"] \arrow[dr, "\sigma"'] & & T \arrow[dl, "\tau"] \\
& A &
\end{tikzcd}$$

The isomophism between σ and τ is written as $\sigma \cong \tau$.

Below, our mechanisation is based on the Agda definition of isomorphisms given in *Programming Language Foundations in Agda* (Wadler et al. [2022]), where `to` and `from` are mappings between `E` and `F`, while `from∘to` and `to∘from` assert `from` is the left-inverse of `to` and vice versa.

We begin with mechanising $\phi : S \cong T$, the isomorphism of event structures:

```
record _≅ₑ_ {Es Fs : Set} (E : EventStructure Es)
                          (F : EventStructure Fs) : Set where
  field
    to : E ↦ F
    from : F ↦ E
    from∘to : ∀ {x : Es} → σ-map from (σ-map to x) ≡ x
    to∘from : ∀ {x : Fs} → σ-map to (σ-map from x) ≡ x
```

Here, `from` and `to` are defined using the total maps between event structures given in Definition 3.1.2, and σ-`map` is the name of the field containing the mapping itself in the ↦ record.

We then wrap _$\cong_e$_ around another record _$\cong$_ to encode isomorphism between pre-strategies σ and τ:

```
record _≅_ {Sₛ Tₛ Aₛ : Set} {S : ESP Sₛ} {T : ESP Tₛ} {A : ESP Aₛ}
              (s : pre-strat S A) (t : pre-strat T A) : Set where
  field
    es-≅ : (event-structure S) ≅ₑ (event-structure T)
```

4.3 Strategies

As stated previously, the guiding principle employed by CCRW in their construction of concurrent strategies is that strategies should act as morphisms in a (bi)category where the objects are games. Therefore, strategies are pre-strategies that are

1. Compositional, and
2. The copycat pre-strategy acts as the compositional identity.

For clarity, we illustrate how strategies are built from the constructions in CCRW through the following dependency graph.

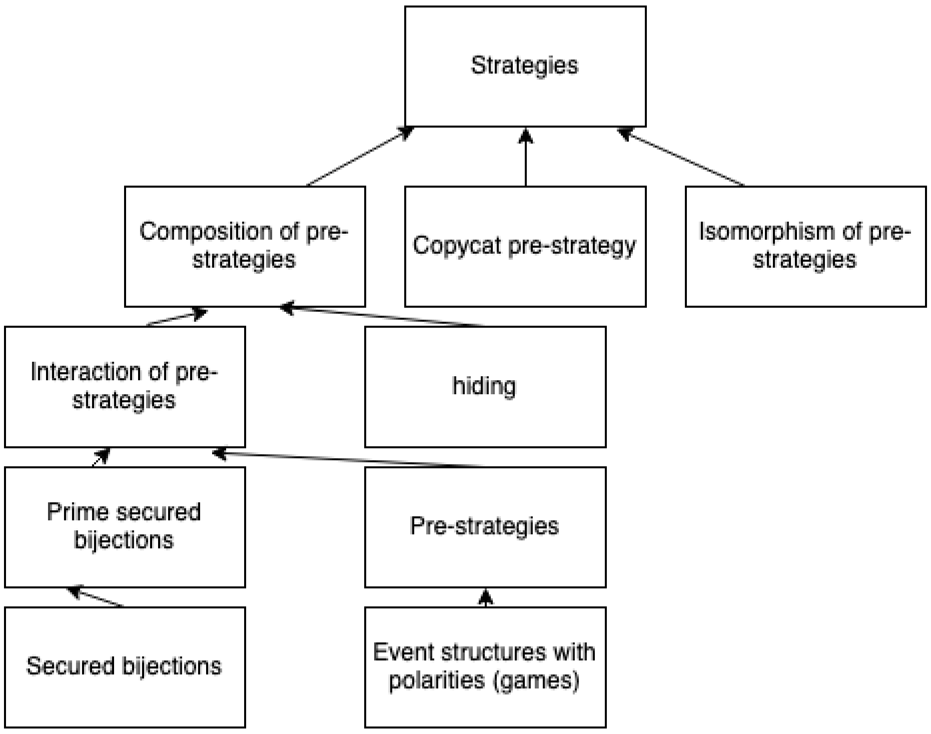
Strategies
Composition of pre-strategies
Copycat pre-strategy
Isomorphism of pre-strategies
Interaction of pre-strategies
hiding
Prime secured bijections
Pre-strategies
Secured bijections
Event structures with polarities (games)

Figure 1: Dependency graph for the construction of strategies.

We have established compositionality for pre-strategies mapping onto the same game, a definition for the copycat pre-strategy, and isomorphism on pre-strategies. All that remains is to combine these three ingredients.

Definition 4.3.1 (CCRW). *(Strategies). A strategy on a game A is a pre-strategy $\sigma : S \rightarrow A$ satisfying:*

$$\mathbf{C}^A \odot \sigma \cong \sigma$$

We are one step away from mechanising Definition 4.3.1. We refer back to our mechanisation of $\odot$, which has type:

```
_⊙_ : ∀ {Sₛ Tₛ Aₛ Bₛ Cₛ : Set} {S : ESP Sₛ} {T : ESP Tₛ}
  {A : ESP Aₛ} {B : ESP Bₛ} {C : ESP Cₛ}
  → (τ : pre-strat T ((Dual B) ∥ C))
  → (σ : pre-strat S ((Dual A) ∥ B))
  → pre-strat (T⊙S τ σ) ((Dual A) ∥ C)
_⊙_ σ τ = polarise
            {!((mapping-σ σ τ) ⊙-map (τ-assoc (mapping-τ σ τ)))!}
```

To write $\mathbf{C}^A \odot \sigma$ in Agda, σ must satisfy this interface where it has type `pre-strat S ((Dual A) ∥ B))`, such that it maps onto an ESP that is the composition of two smaller ESPs. However, we would like to define strategies on the more general class of pre-strategies that have type `pre-strat S A`.

This problem can be resolved by defining an *empty event structure* and corresponding ESP, with no effect on the ESP it is composed with except for making our mechanisation of strategies typecheck. To this end, we define an empty datatype with no consturctors to use as the events of this ESP:

```
data Empty : Set where
```

We can then construct an event structure and an associated ESP, each with no events:

```
es-∅ : EventStructure Empty
es-∅ = record
          { fpo = fpo-∅
          ; Con = con-∅
          }

esp-∅ : ESP Empty
esp-∅ = mkESP (λ x → +) es-∅
```

We assign an arbitrary polarity relation to `esp-∅` because the relation will never be used, as there are no events to apply it to.

Now, we mechanise strategies through the following record:

```
record Strategy {Aₛ Sₛ : Set}
                (A : ESP Aₛ)
                (S : ESP Sₛ) : Set₁ where
  field
    σ : pre-strat S (esp-∅ ∥ A)
    σ-req : (copycat-strat A ⊙ σ) ≅ σ
```

A `Strategy` is constructed from two ESPs `S` and `A`, corresponding to the arguments to the pre-strategy σ in Definition 4.3.1. The σ field contains a pre-strategy from `S` to `A` where `A` is padded with the empty event structure. Then, the σ`-req` field enforces σ to be invariant under composition with the copycat pre-strategy on `A`.

Because our mechanisation of the $\odot$ operation at the end of Chapter 3 has an incomplete type signature, Agda is unable to typecheck σ`-req`. The source code submitted alongside this project employs the following placeholder for $\odot$:

```
_⊙₂_ : ∀ {Sₛ Tₛ Aₛ Bₛ Cₛ Dₛ : Set} {S : ESP Sₛ} {T : ESP Tₛ}
  {A : ESP Aₛ} {B : ESP Bₛ} {C : ESP Cₛ} {D : ESP Dₛ}
  → (τ : pre-strat T ((Dual B) || C))
  → (σ : pre-strat S ((Dual A) || B))
  → pre-strat D ((Dual A) || C)
_⊙₂_ = {!!}
```

The domain of $\odot_2$ is an arbitrary ESP `D`, while the domain of $\odot$ is the set of ESPs satisfying $T \odot S$. Since $\odot_2$ is more general version of $\odot$ and Agda is able to typecheck σ`-req` with $\odot_2$, we are confident it will typecheck for $\odot$ if its type signature is completed.

We now have a mostly complete mechanisation of the constructions needed for expressing a concurrent strategy in Agda.

Chapter 5

Conclusions

5.1 Summary

To summarise, this project provides a partial mechanisation for concurrent games and strategies as constructed in CCRW, which involved developing libraries for finite sets, event structures, and pre-strategies in Agda. This report provides an abridged version of the theory introduced in CCRW, and explains the design choices we made in our implementation work.

5.2 Limitations

We now address the limitations of our project:

- *Incomplete proofs*: The most glaring limitation of our project is the fact that several proofs are left incomplete, as a trade-off for reaching the construction for strategies within the time constraint of this project. We made this decision because the omitted proofs mostly pertain to properties of lists, and are generally irrelevant to the primary aim of this project. However, we recognise that a mechanisation project is only complete when there is nothing left unproven, and we are confident all of the incomplete proofs can be done given additional time.
- *Postulates*: As addressed in subsection 3.3.3, we employed a postulate in place of implementing a library for finite sets that is not based on lists. The postulate will be eliminated once we implement this library.
- *Agda code compilation*: As addressed at the end of Chapter 3, we encountered a problematic case for Agda's unification engine and were forced leave some definitions incomplete. We plan to report this case to the Agda development team.

5.3 Future work

Now, we consider the ways in which our work in this project can be expanded upon in the future.

The most obvious task is to complete the unfinished proofs in the existing parts of this project. There is little to remark about this except that we expect these proofs to be very long and time-consuming. This will also involve implementing a better library for finite sets that either enforces elements to be equatable by Agda's propositional equality, or modifying our implementation of event structures and its associated proofs to use a different equality relation altogether.

Upon completion of the mechanisation work described in this report, additional mechanisations of the categories $\mathcal{E}$, $\mathcal{EP}$, and the bi-category of games through Agda's existing category theory library will provide a fuller mechanisation of CCRW, and improve confidence in the correctness of the constructions in CCRW.

Another promising direction for the project is additional mechanisations for strategies characterised through *receptivity* and *courtesy* (CCRW), and a proof showing this characterisation is equivalent to Definition 4.3.1., which is the main result of Rideau and Winskel [2011].

Bibliography

Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

Simon Castellan, Pierre Clairambault, and Glynn Winskel. The parallel intensionally fully abstract games model of pcf. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 232–243, 2015. doi: 10.1109/LICS.2015.31.

Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. Games and Strategies as Event Structures. *Logical Methods in Computer Science*, September 2017. doi: 10.23638/LMCS-13(3:35)2017. URL https://hal.science/hal-01302713.

Martin Churchill and James Laird. A logic of sequentiality. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 215–229, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15205-4.

Pierre Clairambault and Marc de Visme. Full abstraction for the quantum lambda-calculus. 4(POPL), dec 2019. doi: 10.1145/3371131. URL https://doi.org/10.1145/3371131.

Pierre Clairambault and Glynn Winskel. On concurrent games with payoff. *Electronic Notes in Theoretical Computer Science*, 298:71–92, 2013. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2013.09.008. URL https://www.sciencedirect.com/science/article/pii/S1571066113000546. Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX.

Pierre Clairambault, Julian Gutierrez, and Glynn Winskel. The winning ways of concurrent games. LICS '12, page 235–244, USA, 2012. IEEE Computer Society. ISBN 9780769547695. doi: 10.1109/LICS.2012.34. URL https://doi.org/10.1109/LICS.2012.34.

Andrzej Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*, 49(2):129–141, 1961. URL http://eudml.org/doc/213582.

Jaakko Hintikka and Gabriel Sandu. Game-theoretical semantics. In *Handbook of logic and language*, pages 361–410. Elsevier, 1997.

J.M.E. Hyland and C.-H.L. Ong. On full abstraction for pcf: I, ii, and iii. *Information and Computation*, 163(2):285–408, 2000. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.2000.2917. URL https://www.sciencedirect.com/science/article/pii/S0890540100929171.

Martin Hyland. *Game Semantics*, page 131–184. Publications of the Newton Institute. Cambridge University Press, 1997. doi: 10.1017/CBO9780511526619.005.

Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong. Innocent game models of untyped λ-calculus. *Theoretical Computer Science*, 272(1):247–292, 2002. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(00)00353-4. URL https://www.sciencedirect.com/science/article/pii/S0304397500003534. Theories of Types and Proofs 1997.

Tom Leinster. Basic category theory, 2016.

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

Andrzej S. Murawski and Nikos Tzevelekos. An invitation to game semantics. *ACM SIGLOG News*, 3(2):56–67, may 2016. doi: 10.1145/2948896.2948902. URL https://doi.org/10.1145/2948896.2948902.

Ulf Norell. *Dependently Typed Programming in Agda*, pages 230–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04652-0. doi: 10.1007/978-3-642-04652-0_5. URL https://doi.org/10.1007/978-3-642-04652-0_5.

Silvain Rideau and Glynn Winskel. Concurrent strategies. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 409–418, 2011. doi: 10.1109/LICS.2011.13.

E. A. Moiseenko A. A. Trunov V. P. Gladstein, D. V. Mikhailovskii. *Mechanized theory of event structures: a case of parallel register machine*. PhD thesis, 2021.

Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL https://plfa.inf.ed.ac.uk/22.08/.

Glynn Winskel. Event structures. In *Advances in Petri Nets 1986*. Springer Lecture Notes in Computer Science 255, 1987.