YuugiLang: a Language for Game Semantics

Tarso Boudet Caldas
Master in Computer Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

YuugiLang: a Language for Game Semantics

Tarso Boudet Caldas

Dissertation carried out as part of the Master in Computer Science

Department of Computer Science

2025

Supervisor

Sandra Alves, Assistant Professor
Faculty of Sciences of the University of Porto

Aknowledgements

I would like to express our sincere gratitude to Professor Sandra Alves for her invaluable guidance and support throughout the development of this dissertation. Her expertise in foundations of programming languages has been instrumental in shaping the direction of this work.

I am also thankful to my university colleagues, in particular, Rodrigo Marques and Guilherme Duarte, for their feedback and encouragement during the course of this research.

Finally, I wish to thank my family and friends for their constant support and understanding during this journey. Having them by my side has been a source of strength and motivation. Withouth it, this work would not have been possible.

Abstract

Theoretical models, such as λ-calculus, π-calculus and linear logic, have shaped the way we understand programming languages. Game semantics is a relatively recent approach to formal semantics, but provides a powerful framework to for the interpretation of programs as interactive strategies, connecting logic and computation. Despite its ever growing presence in the academic world, game semantics has not yet been widely adopted in practical programming language design.

This dissertation presents YuugiLang, a minimal programming language designed as a tool for the study of game semantics. The language is equipped with two layers, a *payload layer*, which provides a simple functional programming environment based on the simply typed lambda calculus, and a *process layer*, based on the session-typed π-calculus, that provides constructs for the definition of interactive processes. The payload layer is used to define data to be communicated, while the process layer enforces structure communication over channels. Game semantics is used to provide a denotational semantics for the process layer, interpreting session types as games and processes as strategies between two players.

The implementation is written in Rust. Programs are parsed from source code into an abstract syntax tree, which is then type checked and evaluated by an interpreter that executes both layers.

While the system is not intended as a fully fledged programming language, it serves as a demonstration of how abstract semantics can be applied to the construction of a programming language. We aim here to provide a tool for the study of game semantics, concurrent programming and the implementation of programming languages.

Resumo

Modelos teóricos, como o λ-calculus, o π-calculus e a lógica linear, moldaram a forma como entendemos as linguagens de programação. A semântica de jogos é uma abordagem relativamente recente para a semântica formal, mas fornece um poderoso arcabouço para a interpretação de programas como estratégias interativas, conectando lógica e computação. Apesar de sua presença cada vez maior no meio acadêmico, a semântica de jogos ainda não foi amplamente adotada no *design* prático de linguagens de programação.

Esta dissertação apresenta YuugiLang, uma linguagem de programação mínima concebida como uma ferramenta para o estudo da semântica de jogos. A linguagem é estruturada em duas camadas: uma *camada de dados*, que oferece um ambiente funcional simples baseado no cálculo lambda simplesmente tipado, e uma *camada de processos*, baseada no π-calculus tipado por sessões, que provê construtores para a definição de processos interativos. A camada de dados é usada para definir a informação a ser comunicada, enquanto a camada de processos impõe uma comunicação estruturada sobre canais. A semântica de jogos é utilizada para fornecer uma semântica denotacional para a camada de processos, interpretando tipos de sessão como jogos e processos como estratégias entre dois jogadores.

A implementação foi desenvolvida em Rust. Programas são analisados a partir do código-fonte em uma árvore sintática abstrata, que é então verificada por tipos e avaliada através de um interpretador que executa ambas as camadas.

Embora o sistema não tenha sido concebido como uma linguagem de programação completa, ele serve como uma demonstração de como a semântica abstrata pode ser aplicada à construção de uma linguagem de programação. Nosso objetivo é fornecer uma ferramenta para o estudo da semântica de jogos, da teoria da programação concorrente e da implementação de linguagens de programação.

Contents

List of Figures

Acronyms

ML$_{||}$ Concurrent ML

AST abstract syntax tree

CL classical logic

DAG directed acyclic graph

EBNF extended Backus-Naur form

ES event structure

IL intuitionistic logic

LL linear logic

LTS labelled transition system

PCF programming computable functions

STLC simply-typed λ-calculus

Chapter 1
Introduction

The field of programming languages is a rich and diverse area of study, with many different approaches to the design and implementation of languages. Theoretical frameworks such as the λ-calculus, the π-calculus and linear logic have been the source of inspiration for the development of many modern programming languages. However, it is often challenging to understand how these theoretical concepts can be applied in practice, and how they can be used effectively implemented in a real-world programming language.

In the last few decades, the field of game semantics has emerged as a fully abstract model for functional programming languages, such as the case for programming computable functions (PCF) (J. M. E. Hyland & C.-H. L. Ong, 2000) and a as a bridge between linear logic (LL) and the π-calculus (Simon Castellan et al., 2020; John Laird, 2005; Kenta Sakayori & Nobuko Yoshida, 2017). It also has applications in program analysis and verification (Dan R. Ghica, 2009), making it a powerful tool for understanding the behavior of concurrent programs.

Although the theory of game semantics is ever evolving, there is still a lack of practical implementations of programming languages based on this framework. This work aims to fill this gap by exploring the implementation of a concurrent programming language based on game semantics, serving as a proof of concept for its real-world programming application, and to serve as an educational tool for both those interested in game semantics and those interested in the design and implementation of programming languages.

We call the language we implemented YuugiLang, name derived from the Japanese word "yūgi" (遊戯), which means "game" or "play," reflecting the game's semantics foundation of the language. It is built on two layers: a *payload layer* layer, which is a simple call-by-value functional language based on simply typed λ-calculus, with base types and references. These terms are used as the values that can be exchanged between processes. The second layer is the *process layer*, which is based on a session-typed π-calculus. It uses channels to to enforce structured communication between processes, and to ensure that the communication follows a predefined protocol. This design choice is made mostly based on the articles Castellan et al. (2020) and Sakayori and Yoshida (2017).

Game semantics is used to provide a denotational semantics for the process layer, where programs are interpreted as strategies in a game between two players, with the session types serving as possible moves in the game. This allows us to reason about the behavior of programs in a compositional way, and to ensure that the communication between processes is well-structured and follows predetermined protocols. By constructing a language and interpreter with these ideas, we can make abstract notions of strategies, duality and interaction more concrete, and understand how they can be used in practice.

The implementation is written in Rust, a modern system programming language, and we use an interpreter approach to execute YuugiLang programs. We parse programs into an abstract syntax tree (AST), which is then type-checked, and finally evaluated. Rust, with its strong emphasis on safety and concurrency, is a natural fit for this project, as it allows us to leverage its features to ensure that our

implementation is both safe and efficient, as Rust is known for being a high performance low-level language comparable to C++ (Nikolay Ivanov, 2022).

Scope and limitations. This work does not aim to introduce new theoretical concepts or to prove new results in the field of game semantics. Instead, it focuses on the practical implementation of a minimal programming language based on existing theory. The system is not intended to be a full-featured implementation, but rather a proof of concept that demonstrates the feasibility of using game semantics as a foundation for a concurrent programming language. Its value lies in its ability to illustrate that concepts of game semantics can be mapped step by step in a working language.

The hope is that this work will serve as a starting point for students and researchers interested in both game sematics theory and the design and implementation of programming languages, and that it will inspire further exploration and development in this area.

1.1 Outline

We begin in Chapter 2 by providing the necessary theoretical background and context for understanding game semantics and our implementation. We cover the basics of linear logic, the version of the λ-calculus we use as the payload layer, and the session-typed π-calculus that composes the process layer. The last section of this chapter presents a quick introduction to Rust and its key features, and why it was chosen as the implementation language.

Next, in Chapter 3, we introduce the main concepts of game semantics, including arenas, strategies, and the interpretation of types and terms. We also discuss how game semantics can be used to model concurrency and communication between processes, and how it relates to linear logic and the π-calculus.

The core of this work is presented in Chapter 4 and Chapter 5. In Chapter 4, we present the syntax and static semantics of YuugiLang, including the the rules for type-checking programs and the process of parsing the source code. We discuss the design choices made in the language, and how they relate to the underlying theory of game semantics. Chapter 5 covers the dynamic semantics of YuugiLang, including the evaluation rules and the implementation of the interpreter. Finally, in Chapter 6, we reflect of the lessons learned from this project, discuss its limitations, and suggest directions for future extensions and improvements.

The source code for YuugiLang can be found in Appendix A, and its latest version is also available online at https://github.com/tarsobcaldas/yuugilang.

Chapter 2
Driving Concepts

In this chapter we will present the main theoretical concepts that are the basis for understanding the design and implementation of YuugiLang. Here, we will cover the simply-typed lambda calculus (J. Roger Hindley, 1997), the foundation for the payload layer, linear logic (Jean-Yves Girard, 1987), which relates to the resource management system of the language, and the session-typed π-calculus (Robin Milner, 1999; Vasco T. Vasconcelos, 2009), that is the basis for the process layer and the concurrency model. Finally, we will present a brief introduction to Rust, the implementation language used for the interpreter of YuugiLang.

2.1 Simply typed λ-calculus

The λ-calculus is a formal system for defining and manipulating functions. It is a foundation for functional programming languages such as Haskell and OCaml, and it has deep connections to logic and type theory through the Curry-Howard correspondence. In this section, we introduce the call-by-value simply-typed λ-calculus (STLC), which is the core computational model for the *payload layer* of YuugiLang. This will later be extended into the $\text{ML}_{||}$ (concurrent ML) language as presented by Castellan et al. (2020), which adds mutable state, general recursion, constants and other features to the base λ-calculus. For a more detailed presentation of the λ-calculus, we refer the reader to Henk P. Barendregt (1984), but our presentation follows closely that of Mads Sørensen and Piotr Urzyczyn (2006).

2.1.1 Syntax

We express computation in the λ-calculus through *terms*, which are built from *variables*, *abstractions*, and *applications*. These can have different *types*, which classify the kinds of values that terms can take. The syntax for types is

$$\sigma, \tau ::= \sigma \to \tau$$

jWe assign types to term using *type assignments* of the form $M : \sigma$, where M is a term and σ is its type. A type assignment of the form $M : \sigma \to \tau$ means that the term M is a function that takes an input of type σ and returns a value of type τ.

Definition 2.1 (Sørensen & Urzyczyn, 2006, pp. 63–64)**.**

i) Raw terms are defined by the following rules:

- every variable is a raw term;
- if M, N are raw terms, then (MN) is a raw term;

- if x is a variable and M is a raw term, then $(\lambda x{:}\sigma.M)$ is a raw term;

ii) *Free variables* of a raw term M is defined by

$$\begin{aligned} \mathrm{fv}(x) &= \{x\} \\ \mathrm{fv}(\lambda x{:}\sigma.M) &= \mathrm{fv}(M) \setminus \{x\} \\ \mathrm{fv}(MN) &= \mathrm{fv}(M) \cup \mathrm{fv}(N) \end{aligned}$$

If $\mathrm{fv}(M) = \varnothing$, then M is a *closed term*.

iii) A variable x is considered *bound* in the term $\lambda x{:}\sigma.M$.

Example 2.1.

- x is a raw term with $\mathrm{fv}(x) = \{x\}$.
- If $M = \lambda x{:}\sigma.x$ then M is a raw term and $\mathrm{fv}(M) = \varnothing$ (since x is bound).
- If $M = (\lambda x{:}\sigma.x)y$ then M is a raw term and $\mathrm{fv}(M) = \{y\}$.
- If $M = \lambda x{:}\sigma.xy$ then $\mathrm{fv}(M) = \{y\}$ (x is bound, y is free).

∘

We adopt the Barendregt convention (Barendregt, 1984, p. 26), which states that bound variables are chosen to be different from free variables. This allows us to avoid renaming bound variables when performing substitutions. We also omit parentheses whenever doing so does not lead to ambiguity — $\lambda x{:}\sigma.xy$ is shorthand for $(\lambda x{:}\sigma.(xy))$.

Definition 2.2 (Sørensen & Urzyczyn, 2006, p. 64)**.** The substitution of a raw term N for a variable x in M, written $M\{N/x\}$, is defined as follows:

$$\begin{aligned} x\{N/x\} &= N \\ x'\{N/x\} &= x', \text{ if } x' \neq x \\ (PQ)\{N/x\} &= P\{N/x\}Q\{N/x\} \\ (\lambda y{:}\sigma.P)\{N/x\} &= \lambda y{:}\sigma.P\{N/x\}, \text{ where } y \neq x \text{ and } y \notin \mathrm{fv}(N) \end{aligned}$$

The condition $y \notin \mathrm{fv}(N)$ in the last case is necessary to avoid *variable capture*, which occurs when a free variable in N becomes bound after the substitution. If this condition is not met, we can rename y to a fresh variable that does not appear in N or M before performing the substitution.

Example 2.2. Let $M = \lambda x{:}\sigma.xy$ and substitute y for $\lambda z{:}\tau.z$:

$$(\lambda x{:}\sigma.xy)\{\lambda z{:}\tau.z/y\} = \lambda x{:}\sigma.x(\lambda z{:}\tau.z).$$

No capture occurs since the substituted free variables do not coincide with x. ∘

Reduction. A raw term M can be reduced to another raw term N by applying a reduction rule. The most basic reduction rule is the *beta-reduction*.

Definition 2.3. The relation $\to_\beta$ is defined as the smallest relation such that

$$(\lambda x{:}\sigma.M)v \to_\beta M\{v/x\},$$

satisfying the following rules:

i) If $M \to_\beta N$, then $\lambda x{:}\sigma.M \to_\beta \lambda x{:}\sigma.N$

ii) If $M \to_\beta N$, then $MM' \to_\beta NM'$

iii) If $M \to_\beta N$, then $M'M \to_\beta M'N$

Example 2.3. Let M be the term $(\lambda x{:}\sigma.xx)(\lambda y{:}\sigma.y)$. Then, we have

$$M \to_\beta (\lambda y{:}\sigma.y)(\lambda y{:}\sigma.y).$$

Continuing, we get

$$(\lambda y{:}\sigma.y)(\lambda y{:}\sigma.y) \to_\beta \lambda y{:}\sigma.y,$$

in which no further reductions are possible. ∘

2.1.2 Typing Rules

Having a type system allows us to reason about the behavior of programs before executing, as well as to detect errors early in the development process. We adopt a *Church-style* type system, where type annotations are written explicitly on λ-abstractions. This means that we do not require inference, as the types are already provided in the syntax of the terms. In contrast, a *Curry-style* type system does not require type annotations, and types are inferred from the structure of the terms. This choice is motivated by the fact that our main goal is to provide a clear and formal account of the semantics of the language, and not to focus on type inference algorithms.

A *type judgement* is of the form $\Gamma \vdash M{:}\sigma$, where Γ, the context or *type environment*, is a map $\{x_1{:}\sigma_1, \dots, x_n{:}\sigma_n\}$ from variables $x_1, \dots, x_n$ to types $\sigma_1, \dots, \sigma_n$. This means that $x{:}\sigma \in \Gamma$ can be understood as a function $\Gamma : \mathrm{dom}(\Gamma) \to \mathrm{rg}(\Gamma)$, where $\mathrm{dom}(\Gamma) = \{x_1, \dots, x_n\}$ and $\mathrm{rg}(\Gamma) = \{\sigma_1, \dots, \sigma_n\}$.

If we have $\Gamma \cap \Gamma' = \varnothing$, then we write Γ, Γ' for the union of the two contexts, that is $\Gamma \cup \Gamma'$. Notably, $\Gamma, x{:}\sigma$ stands for $\Gamma \cup \{x{:}\sigma\}$, and is required that x does not appear in Γ $x \notin \mathrm{dom}(\Gamma)$. The typing rules are presented in Table 1.

Definition 2.4 (Sørensen & Urzyczyn, 2006, p. 64)**.** We say that M is a *term* of type σ in Γ, and write $\Gamma \vdash M{:}\sigma$, when $\Gamma \vdash M{:}\sigma$ can be derived using the rules in Table 1.

Example 2.4 (Sørensen & Urzyczyn, 2006, p. 57)**.** Notable examples of λ-abstractions are the combinators **I** (identity), **K** (constant), and **S** (substitution):

$$\begin{aligned}\mathbf{I} &= \lambda x{:}\sigma.x,\\ \mathbf{K} &= \lambda x{:}\sigma.\lambda y{:}\tau.x,\\ \mathbf{S} &= \lambda x{:}(\sigma \to \tau \to \rho).\lambda y{:}(\sigma \to \tau).\lambda z{:}\sigma.xz(yz).\end{aligned}$$

Their respective typing judgements are:

i) $\vdash \mathbf{I}{:}\sigma \to \sigma$

ii) $\vdash \mathbf{K}{:}\sigma \to \tau \to \sigma$

iii) $\vdash \mathbf{S}{:}(\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$

∘

Table 1
Typing rules for simply-typed λ-calculus.

$$\frac{}{\Gamma, x{:}\sigma \vdash x{:}\sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash M{:}\sigma \to \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau}\ \textsc{App}$$

Normalization. A term M is said to be *strongly normalizing* if there are no infinite sequences of reductions starting from M. In other words, every sequence of reductions starting from M eventually reaches a normal form. A term is in *normal form* if it cannot be further reduced .

In the simply-typed λ-calculus, all well-typed terms are strongly normalizing. This no longer holds when we add features like general recursion or mutable state, as they can lead to non-terminating computations, such as infinite loops, as we will see in the next section.

2.1.3 Concurrent ML

With the fundamentals of the STLC in place, we can now extend it in order to obtain a more practical programming language. We will follow the presentation of Castellan et al. (2020), who defines $\text{ML}_{||}$ (concurrent ML) as concurrent call-by-value language with integer references. The syntax is as shown in Table 2.

Table 2
Grammar for $\text{ML}_{||}$. From Castellan et al. (2020, p. 3)

type	$\sigma, \tau ::= \bullet \mid \texttt{int} \mid \texttt{bool} \mid \texttt{alloc} \mid \sigma \to \tau$	value	$v ::= x \mid \lambda x{:}\sigma.M \mid \underline{n} \mid \texttt{true} \mid \texttt{false} \mid ()$
term	$M, N ::= v \mid MN \mid Y \mid \bot \mid \texttt{if}\, MNN' \mid \texttt{plus} \mid \texttt{equal} \mid \texttt{alloc} \mid \texttt{get} \mid \texttt{set}$		

Values. Beyond types and terms, we now also include *values*, which are terms that are fully evaluated and cannot be reduced further. Values include variables, λ-abstractions, numeric constants, boolean constants (`true`, `false`), and the unit value (). Although we represent generic numeric constants as $\underline{n}$ in the grammar, we will use standard natural numbers $0, 1, 2, \ldots$ in examples for clarity.

Types. We now add new types to the language, namely the $\bullet$ (unit type), `int` (integer type), `bool` (boolean type), and `alloc` (reference type). The unit type has a single value, which is (), representing the absence of a meaningful value. The integer type represents whole numbers, while the boolean type represents truth values, `true` and `false`. The reference type is used to represent mutable state, allowing us to create variables that can be updated.

Terms. Along with application, abstraction, and variables — the last two being values — from the STLC, we add several new terms to the language. These include:

- $\texttt{if}\, MNN'$: a conditional expression that evaluates M and, if it returns `true`, evaluates N; otherwise, it evaluates N'.
- Y: a fixed-point combinator that allows us to define recursive functions. It satisfies $Y\, f = f\,(Y\, f)$.

- `plus` and `equal`: built-in functions for addition and equality comparison of integers, respectively.
- `alloc`, `get`, and `set`: terms for working with mutable references. `alloc` creates a new reference, `get` retrieves the value stored in a reference, and `set` updates the value of a reference.
- $\bot$: represents a computation that does not terminate or results in an error.

The terms `plus`, `equal`, `alloc`, `get`, and `set` are treated as *higher-order constants*, meaning that they can be applied to arguments like regular functions. We write $M + N$ for $\texttt{plus}\,MN$ and $M = N$ for $\texttt{equal}\,MN$. In the case of the mutable references, we write $M := N$ for `set`, where M is a reference and N is the new value to be set, while `alloc` and `get` are used as is. The typing rules for these terms follow the same principles of Section 2.1.2, and are presented in Table 3.

Table 3
Typing rules for $ML_{||}$. From Castellan et al. (2020, p. 4)

$$\frac{}{\Gamma, x{:}\sigma \vdash x{:}\sigma}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash \bot{:}()}\ \textsc{Bot}$$

$$\frac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash M{:}\sigma \to \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau}\ \textsc{App}$$

$$\frac{}{\Gamma \vdash Y{:}((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau}\ \textsc{Fix} \qquad \frac{}{\Gamma \vdash \texttt{true}, \texttt{false}{:}\texttt{bool}}\ \textsc{Bool}$$

$$\frac{\Gamma \vdash M{:}\texttt{bool} \quad \Gamma \vdash N_1{:}\sigma \quad \Gamma \vdash N_2{:}\sigma}{\Gamma \vdash \texttt{if}\,MN_1N_2{:}\sigma}\ \textsc{If} \qquad \frac{}{\Gamma \vdash \underline{n}{:}\texttt{int}}\ \textsc{Num}$$

$$\frac{}{\Gamma \vdash \texttt{plus}{:}\texttt{int} \to \texttt{int} \to \texttt{int}}\ \textsc{Plus} \qquad \frac{}{\Gamma \vdash \texttt{equal}{:}\texttt{int} \to \texttt{int} \to \texttt{bool}}\ \textsc{Equal}$$

$$\frac{}{\Gamma \vdash (){:}\bullet}\ \textsc{Unit} \qquad \frac{}{\Gamma \vdash \texttt{alloc}{:}\texttt{int} \to \text{ref}}\ \textsc{Alloc}$$

$$\frac{}{\Gamma \vdash \texttt{get}{:}\ \text{ref} \to \texttt{int}}\ \textsc{Get} \qquad \frac{}{\Gamma \vdash \texttt{set}{:}\ \text{ref} \to \texttt{int} \to \bullet}\ \textsc{Set}$$

We will not present the operational semantics of $ML_{||}$ here, as we need to first introduce the process layer and the concurrency model in Section 2.3. We will come back for these details in Chapter 5, where we present the full operational semantics of YuugiLang.

2.2 Linear Logic

Linear logic (LL) is a refinement of traditional logics, such as classical logic (CL) and intuitionistic logic (IL), introduced by Girard (1987). It differs from these logics by its treatment of propositions as resources, rather than as truths that can be used freely. This means that in LL, we cannot duplicate or discard propositions without explicitly stating so. In this section, we will present the fragments of LL that are relevant for connecting it to game semantics and the *session-typed* π-calculus. Our presentation follows linear logic as described by Maribel Fernández and Ian Mackie (2002) and Philip Wadler (1993).

2.2.1 Linear Logic vs. Traditional Logic

While both CL and IL focus on the notion of pursuing truth, LL emphasizes the management of *resources*. This idea derives from the notion that not everything is free. For example, in traditional

logic, we can reuse assumptions as many times as we want, and we can also discard them if we do not need them anymore. When we use a fact to conclude another, we still have it available for future use.

Example 2.5. Wadler (1993, p. 1) Consider the case

$$A \to B, A \vdash A \times B.$$

We can read this as: given the assumption $A \to B$ and the assumption A, we can deduce A *and* B. Here, we rely on the assumption A twice: once to deduce B and once to conclude the conjunction. ∘

We do not always have unlimited resources. If we are cooking and we have two recipes that both require an egg, but we only have one egg, we cannot prepare both. The same is the case for real computation, where resources such as memory and processing power are limited. Linear logic captures this idea by tracking the usage of assumptions and ensuring that they are not duplicated or discarded without explicit permission.

Proof Systems. We usually use *proof systems* to derive the truth values of propositions. The two most common proof systems are *natural deduction* and *sequent calculus* (Gerhard Gentzen, 1935). In both systems, *sequents* are used to represent the relation between *assumptions* and *conclusions*, which are composed of *propositions*.

A proposition is a formula built from propositional constants and logical connectives. We will use the notation in Wadler (1993), where the logical connectives $\times$, $+$, and $\to$ represent *and*, *or*, and *implies*, respectively. Let A, B, C be propositions and X a propositional constant. Then, we can define propositions by the following grammar:

$$A, B, C ::= X \mid A \times B \mid A + B \mid A \to B$$

Definition 2.5. A *sequent* is a pair $\Gamma \vdash \Delta$, where Γ, Δ are sequences of propositions. We understand the sequent $A_1, \dots, A_m \vdash B_1, \dots, B_n$ as being equivalent to $A_1 \times \cdots \times A_m \vdash B_1 + \cdots + B_n$.

Sequents are taken to mean the provability of the formulae on the right-hand side given the formulae on the left-hand side (usually called the *context*). In natural deduction, sequents are more commonly referred to as *judgements*, and have the form $\Gamma \vdash A$ (i.e., a single conclusion).

Definition 2.6. A *proof rule* is a representation of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma' \vdash \Delta'} R,$$

where R is the rule that, when applied, allows us to build a proof of $\Gamma' \vdash \Delta'$ from the proofs of all $\Gamma_i \vdash \Delta_i$.

We can split the proof rules in three categories (Fernández & Mackie, 2002, p. 18):

Identity group The Axiom ($A \vdash A$) and Cut, which allows us to compose proofs. If we have a proof of $\Gamma \vdash A, \Delta$ and a proof of $\Gamma', A \vdash \Delta'$, we can infer $\Gamma, \Gamma' \vdash \Delta, \Delta'$.

Structural group *Exchange* (reordering), *weakening* (discarding), and *contraction* (duplication).

Logical group Introduction and elimination of logical connectives, such as $\times$, $+$, and $\to$.

The main difference between LL and traditional logics lies in the treatment of the structural group. In CL and IL, weakening and contraction are allowed freely, (with some restrictions in IL), while in LL, they are only allowed for propositions marked with the *exponential* modalities ("of course" and "why not"), considered as first-class logical rules (Fernández & Mackie, 2002), which we will introduce later in Section 2.2.3.

The absence of weakening and contraction forces profound changes in how logic connectives must be handled. Implication now becomes a *linear implication*, written $\multimap$ (entails), which $A \multimap B$ can be read as "consume A yielding B" (Wadler, 1993, p. 12). We are also required to differentiate between two kinds of conjunctions and disjunctions, the ones that *share* resources and the ones that *split* them. We call these *additive* and *multiplicative* connectives, respectively, and will be the next focus of discussion.

2.2.2 Multiplicative and Additive Connectives

In traditional logic, whether in the sequent calculus or natural deduction, we can freely manipulate contexts in logical rules. For example, right introduction rule for conjuction ($\times$) in the sequent calculus can be presented in two different ways:

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \times B, \Delta}\, \mathrm{R}\times_{\text{add}} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \times B, \Delta, \Delta'}\, \mathrm{R}\times_{\text{mult}}$$

The first rule shares the same contexts Γ, Δ on both premises, while the second splits them into separate contexts, using Γ, Δ for the first premise and Γ', Δ' for the second. In the additive version, these shared contexts are merged when we conclude the conjunction, while in the multiplicative version, we concatenate the contexts, keeping them separate.

With weakening and contraction available, these two forms are equivalent, as we can discard or duplicate assumptions as needed. However, in LL, this is not the case, as we cannot use these rules freely, so the distinction between *additive* and *multiplicative* connectives becomes essential (Fernández & Mackie, 2002, p. 36).

Therefore, linear logic introduces two pairs of connectives for conjunction and disjunction:

- *additive connectives:* & (with) and $\oplus$ (plus);
- *multiplicative connectives:* $\otimes$ (tensor) and $⅋$ (par).

Each connective has its corresponding unit, also separated into additive and multiplicative units. In Table 4, we summarize the connectives and their units.

Table 4
Linear logic units and connectives

with	&	top (additive true)	$\top$
plus	$\oplus$	nil (additive false)	$\mathbf{0}$
tensor	$\otimes$	unit (multiplicative true)	$\mathbf{1}$
par	$⅋$	bottom (multiplicative false)	$\bot$

Resource interpretation. The rules for linear logic connectives can be understood in terms of combinations of resources (Wadler, 1993, p. 13). For example, consider A as "I have 10 €", B as "I have a ticket to the cinema", and C as "I have a popcorn". Then

$$A \vdash B \qquad A \vdash C$$

can be read as "I can exchange 10 € for a cinema ticket", and "I can exchange 10 € for a popcorn". Combining these through the "tensor" connective, we have

$$\frac{A \vdash B \quad A \vdash C}{A, A \vdash B \otimes C} \otimes\text{–I}$$

which would mean "I can exchange 20 € for a cinema ticket and a popcorn". Here, the contexts are *split*, that is, each premise uses its own copy of the resource, so we must provide both of them. This illustrates that $\otimes$ requires the composition of resources.

For "with", the same resource may reused in both premises:

$$\frac{A \vdash B \quad A \vdash C}{A \vdash B \& C} \&\text{–I}$$

meaning "I can exchange 10 € for either a cinema ticket or a popcorn". The contexts are *shared*, that is, we only need one copy of the resource, but we must choose which of the two propositions we want to obtain. For this reason, & represents an *external choice*, where the environment decides which branch to take.

Dually, the additive disjunction $\oplus$ (*plus*) represents and *internal choice*, where the proof decides which branch to take, committing to one of the options. In the example, this would mean that "I will choose whether to buy a cinema ticket or a popcorn." Finally, $\parr$ (*par*) is the counterpart of $\otimes$, corresponding to a continuation, in which $A \parr B$ means that we first consume A and then B, using independent resources for each.

In summary, the multiplicative connectives ($\otimes$ and $\parr$) *split* resources into independent pairs, while the additive connectives (& and $\oplus$) *share* the same resource, but requires a decision to be made about which proposition to pursue.

Duality. A key concept in linear logic is *duality*, also called *linear negation*, represented by the operator $(\cdot)^\perp$ (read as "perp"). (Fernández & Mackie, 2002, p. 40). Every proposition A has a dual $A^\perp$, which is obtained by reversing the polarities of the connectives. If A is an action that consumes a resource, then $A^\perp$ is an action that produces it. It is also an involutive operator, meaning that $(A^\perp)^\perp = A$. The duality of connectives is defined as follows:

$$\begin{aligned} (A \otimes B)^\perp &= A^\perp \parr B^\perp & \qquad (A \& B)^\perp &= A^\perp \oplus B^\perp \\ (A \parr B)^\perp &= A^\perp \otimes B^\perp & \qquad (A^\perp)^\perp &= A \\ \top^\perp &= \mathbf{0} & \qquad \mathbf{0}^\perp &= \top \\ \mathbf{1}^\perp &= \bot & \qquad \bot^\perp &= \mathbf{1} \end{aligned}$$

The duality of propositions allows us to express the idea of *complementary resources*, where the consumption of a resource in one proposition corresponds to the production of the same resource in its dual. This concept is essential for understanding both the symmetry of communication in session types and the interaction between players in game semantics, which we will explore in Section 2.3.3 and Chapter 3, respectively.

2.2.3 Exponentials and Linear Implication

As we have already mentioned, linear logic excludes the structural rules of weakening and contraction in order to enforce strict control over resource usage. However, there are situations in which we may want to allow for the duplication or discarding of resources. To accommodate this need, Girard (1987) reintroduces these rules through the modalities $!A$ (read as *of course* A) and $?A$ (read as *why not* A), called *exponentials*.

Exponentials. The modal operator $!(\cdot)$ allows us to mark a proposition as reusable, meaning that we can duplicate or discard it as needed. This reintroduces the structural rules of weakening and contraction, but only for the formulas under !. The dual operator $?(\cdot)$ satisfies

$$(!A)^{\perp} = ?(A^{\perp})$$

and allows similar behavior for the dual side of the sequent. We can think of $!A$ as "I have an unlimited supply of A", or as a persistent resource that can be used multiple times without being consumed. On the other hand, $?A$ can be interpreted as "I may choose to use A zero or more times", allowing for optional usage.

The introduction of these modals forms a bridge between linear logic and traditional logics, and permits us to recover familiar reasoning patterns when needed, and to embed classical and intuitionistic logics within the linear logic framework (Wadler, 1993, p. 17).

In the linear sequent calculus, the rules of weakening and contraction are added to a new logical group, called the exponential group, which also adds promotion and dereliction rules for introducing and eliminating the exponentials.

Exponentials are also a source of inspiration for the notion of *replication* in the π-calculus (Luís Caires et al., 2016, p. 367), as Milner (1999) introduced the operator $!P$ to represent processes that can be spawned an unbounded number of times. We will explore this connection further in Section 2.3.

Linear implication. Finally, we can define *linear implication* $\multimap$ (entails), which is a resource-aware version of the traditional implication $\to$. It expresses that a resource A can be transformed into another resource B, consuming it in the process. Linear implication can be defined in terms of the multiplicative connectives as

$$A \multimap B = A^{\perp} \parr B.$$

This is similar to the definition of implication in traditional logic, where $A \to B = \neg A + B$.

Although $\multimap$ is not equivalent to $\to$, we can recover the traditional implication by using the exponential modality:

$$A \to B = (!A) \multimap B$$

Meaning that if we have an unlimited supply of A, we can use it to derive B without consuming it.

With the addition of exponentials and linear implication, we can now present the full grammar of linear logic propositions A, B, where X represents a propositional constant:

$$A, B ::= X \mid A \otimes B \mid A \parr B \mid A \& B \mid A \oplus B \mid A \multimap B \mid \neg A \mid\ !A \mid\ ?A \mid \top \mid \mathbf{0} \mid \mathbf{1} \mid \perp$$

2.3 π-Calculus

The π-calculus is a process calculus that was introduced by Milner et al. (1992) as a formalism for describing concurrent systems. It is based on the idea of processes that can communicate with each other through channels, which are represented as names.

Consider the existence of an infinite set $\mathcal{N}$ of names, which are represented by lowercase letters. Names can be channels, usually denoted by $c, d, \ldots$, or variables, denoted by $x, y, z, \ldots$ — channels can be considered as variables, so there will be cases where we will use the notation for variables as

a channel. We also define *action prefixes* π as either sending or receiving a message, or making a silent transition. This is done by the following grammar:

$$\begin{aligned} \pi ::= {} & c(\boldsymbol{x}) && \text{receive } x \text{ along } c \text{ (input action)} \\ & \overline{c}\langle \boldsymbol{x} \rangle && \text{send } x \text{ along } c \text{ (output action)} \\ & \tau && \text{internal action.} \end{aligned}$$

Following this, we can define processes:

Definition 2.7 (Standard π-calculus, Milner, 1999, p. 87)**.** The set P of process expressions is defined by:

$$P ::= \sum_{i \in I} \pi_i.P_i \;\Big|\; P_1 \,|\, P_2 \;\Big|\; \nu x\, P \;\Big|\; !P.$$

We call processes in the form $\sum_{i \in I} \pi_i.P_i$ — for a finite index set I — *summations* or *sums*. A parallel composition of processes P_1 and P_2 is denoted by $P_1 \,|\, P_2$, while $\nu x\, P$ denotes the restriction of name x in process P. Finally, $!P$ represents the replication of process P.

In a sum, each action π_i must occur before the process P_i, and we say that P_i is *guarded* by π_i. We consider $\mathbf{0}$ as the empty sum, and it is always the final process in a sequence of actions. For example, $c(\boldsymbol{x}).\overline{d}\langle \boldsymbol{x} \rangle.\mathbf{0}$ is a process that receives a message x on channel c, then sends it on channel d, and then does nothing else. For simplicity, we will omit the final $\mathbf{0}$ in processes, so the previous process can be written as $c(\boldsymbol{x}).\overline{d}\langle \boldsymbol{x} \rangle$.

A summation can be seen as an alternative between the processes that form it. For example, in the process $P = c(\boldsymbol{x}).\overline{d}\langle \boldsymbol{x} \rangle + \overline{c}\langle \boldsymbol{y} \rangle$ we have a summation of two processes, one that receives a message x on channel c and then sends it on channel d — call it P_1, and another, that we will name as P_2, that sends a message y on channel c. When executing P, we can either choose to execute P_1 or P_2.

When composing processes, we will adopt the following precedence rules, from highest to lowest:

- Replication ($!P$);
- Restriction ($\nu x\, P$);
- Prefix ($\pi.P$, where π is an input/output action);
- Parallel composition ($P \,|\, Q$);
- Summation ($P + Q$).

Prefix is considered as being right-associative, so $P.Q.R$ is grouped as $P.(Q.R)$, while summation and parallel composition are associative. Parentheses may be used to override precedence. This all means that a process like

$$c(\boldsymbol{x}).\nu y\, P \,|\, \overline{c}\langle \boldsymbol{z} \rangle + Q.!R$$

would be parsed as

$$\left((c(\boldsymbol{x}).(\nu y\, P)) \,|\, \overline{c}\langle \boldsymbol{z} \rangle \right) + (Q.(!R)).$$

We should note as well that when dealing with the restriction of more than one name, it is possible to write $\nu x_1 x_2 \cdots x_n\, P$ instead of $\nu x_1\, \nu x_2 \cdots \nu x_n\, P$, and also use $\vec{x}$ as a shorthand for $x_1, x_2, \ldots, x_n$, so the previous process can also be written as $\nu \vec{x}\, P$.

Names can be either *free* or *bound*. A name is bound if it is introduced either by an input action, or by a restriction. For example, in the process $\nu c\, c(\boldsymbol{x}).P$, both the names c and x are bound, and their scope is limited to process P, that is, the process in the continuation. Any name that is not bound, is considered free.

Definition 2.8. The set of *free names* of a process P, denoted by $\text{fn}(P)$, is recursively defined as follows:

- $\text{fn}(\mathbf{0}) = \varnothing$;
- $\text{fn}\big(\overline{c}\langle \boldsymbol{x} \rangle.P\big) = \{c, x\} \cup \text{fn}(P)$;
- $\text{fn}(c(\boldsymbol{x}).P) = \{c\} \cup \text{fn}(P) \setminus \{x\}$;
- $\text{fn}(P_1 \mid P_2) = \text{fn}(P_1) \cup \text{fn}(P_2)$;
- $\text{fn}(\nu x\, P) = \text{fn}(P) \setminus \{x\}$;
- $\text{fn}(!P) = \text{fn}(P)$.
- $\text{fn}(P + Q) = \text{fn}(P) \cup \text{fn}(Q)$

2.3.1 Labelled Transition System

The behavior of processes in the π-calculus is typically represented via a labelled transition system (LTS), where transitions are written as $P \xrightarrow{\alpha} Q$, meaning that process P can perform an action α and evolve into process Q.

Definition 2.9. A labelled transition system (LTS) is a tuple $(\mathcal{Q}, \Sigma, \mathcal{T})$, where

- $\mathcal{Q}$ is a set of states (processes);
- Σ is a set of actions, with $\tau \notin \Sigma$;
- a relation $\mathcal{T} \subseteq \mathcal{Q} \times \Sigma_\tau \times \mathcal{Q}$ called transition relation, with $\Sigma_\tau = \Sigma \cup \tau$;

The representation of a LTS is a directed graph, where the nodes are the states and the edges are the transitions between states. While similar to a finite automaton, a LTS does not have neither an initial state nor set of final states, as the processes can continue to evolve indefinitely.

Example 2.6. Consider the LTS in Figure 1 with states P, Q, R and the transitions $P \xrightarrow{\overline{c}\langle \boldsymbol{x} \rangle} Q$, $P \xrightarrow{c(\boldsymbol{x})} R$ and $Q \xrightarrow{\tau} R$.

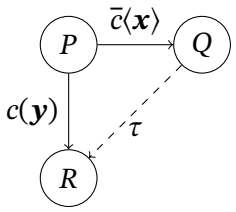

Figure 1
A simple LTS

We can understand this LTS as a program that is either waiting for a message on channel c or sending a message on the same channel. If it sends a message, it will transition to state Q, and then it can perform an internal (non-observable) action to transition to state R. If it receives a message, it will transition directly to state R. ∘

It is important to consider when two LTS are equivalent, as they can exhibit the same observable behavior, even if they are not structurally the same. This is done by understanding the notion of simulation.

Definition 2.10 (Strong Simulation)**.** Let $\mathcal{L}, \mathcal{M}$ be two LTS and $\mathcal{S}$ be a simulation relation between their states. We say that $\mathcal{L}$ *strongly simulates* $\mathcal{M}$ if, for every state $p \in Q$ in $\mathcal{L}$, and for every action $\alpha \in \Sigma$, if $p \xrightarrow{\alpha} p'$ in $\mathcal{L}$, then there exists a state $q \in Q'$ in $\mathcal{M}$ such that $q \xrightarrow{\alpha} q'$ in $\mathcal{M}$ and $p' \mathcal{S} q'$.

For two LTS to be equivalent, they must simulate each other. When that is the case, we call that a *bisimulation* relation.

Definition 2.11 (Strong bisimulation, strong equivalence)**.** We call the binary relation $\mathcal{S}$ a *strong bisimulation* over two LTS $\mathcal{L}$ and $\mathcal{M}$ if both $\mathcal{L}$ and $\mathcal{M}$ strongly simulate each other. The states $p \in Q$ and $q \in Q'$ are *strongly equivalent* or *strongly bisimilar*, written as $p \sim q$ if there exists a strong bisimulation relation $\mathcal{S}$ such that $p \mathcal{S} q$.

In the case of Figure 2, we can see that the LTS $\mathcal{L}$ on the left is not equivalent to $\mathcal{M}$ on the right. Although we could simulate $\mathcal{M}$ with the $\mathcal{L}$, by substituting both q_1, q_1' for p_1, and q_2, q_3 for p_2, p_3, respectively, there is no way to substitute the states of the LTS on the right to simulate the LTS on the left, as both q_1 and q_1' only have a single transition, while p_1 has two transitions. Thus, we say that $\mathcal{L}$ simulates $\mathcal{M}$, but not the other way around, and therefore they are not equivalent.

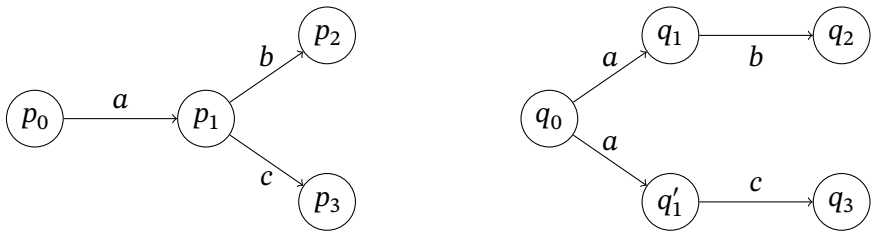

Figure 2
Two LTS that are not equivalent

Example 2.7 (Milner, 1999, p. 19)**.** Consider the LTS in Figure 3. We can see that both LTS are equivalent, as we have $p_0 \sim q_0$. This can be shown by the following relation:

$$\mathcal{S} = \{(p_0, q_0), (p_0, q_2), (p_1, q_1), (p_2, q_1)\}.$$

∘

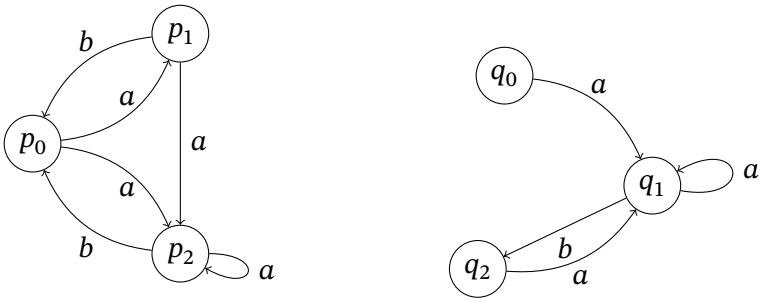

Figure 3
Equivalent LTS

2.3.2 Reactions

In the λ-calculus, we analyze the behavior of programs via reductions, the π-calculus uses a similar concept called *reactions*.

Example 2.8. For motivation, we will show informally how the process of reaction works before defining it properly. Consider the process

$$P = \nu c\left(c(\boldsymbol{x}).\overline{x}\langle \boldsymbol{y}\rangle \,|\, \overline{c}\langle \boldsymbol{d}\rangle\right) | \, d(\boldsymbol{z}).Q.$$

In this example, we have a parallel composition of two processes, one that is inside a restriction and another outside of it. The process inside the restriction is another parallel composition of two processes, one that receives a message on channel c and then sends it on the name x, and another that sends the name d on channel c. The process outside of the restriction receives a message on channel d and then continues as process Q.

First step. We handle the communication inside the restriction, that is, the process $\nu c\left(c(\boldsymbol{x}).\overline{x}\langle \boldsymbol{y}\rangle \,|\, \overline{c}\langle \boldsymbol{d}\rangle\right)$. Here, we have the private channel c with the bound name x in one side of the composition, and on the other side, we have a subprocess that sends the name d through channel c. This means that we can have an interaction between the two processes, and thus we can have a reaction $P \to P'$, where

$$P' = \nu c\left(\overline{d}\langle \boldsymbol{y}\rangle \,|\, \mathbf{0}\right) | \, d(\boldsymbol{z}).Q.$$

This occurs because when we receive a message on channel c, we substitute the name that it carries, x, with the name that was sent. In other terms, we have $\{d/x\}$, which is a substitution of the name x with the name d.

Second step. We can now do the communication in parallel through the channel d — in this case is a public channel. Doing so, we have the reaction $P' \to P''$, with

$$P'' = \nu c\,(\mathbf{0} \,|\, \mathbf{0}) \,|\, Q\{y/z\}.$$

Now we only have the process Q left with the substitution $\{y/z\}$, which means that any occurrence of z should be replaced by y. This is the final step that we can do, as P'' is now in a normal form, meaning that it cannot be reduced any further. ◦

To define reactions, we also need the concept of a structural congruence relation. The latter also requires us to define process context and process congruence.

Definition 2.12 (Milner, 1999, p. 89)**.** *Process contexts* $\mathcal{C}$ are denoted by the grammar

$$\mathcal{C} ::= [\,] \;\Big|\; \pi.\mathcal{C} + P \;\Big|\; \nu x\, \mathcal{C} \;\Big|\; \mathcal{C} \,|\, P \;\Big|\; P \,|\, \mathcal{C} \;\Big|\; !\mathcal{C}.$$

Writing $\mathcal{C}[P]$ means the result of filling $[\,]$ in context $\mathcal{C}$ with process P. We call $\pi.[\,] + P$, $\nu x\,[\,]$, $[\,] \,|\, P$, $P \,|\, [\,]$ and $![\,]$ *elementary contexts*.

Example 2.9. Consider the process context $\mathcal{C} = c(\boldsymbol{x}).[\,] \,|\, \overline{c}\langle \boldsymbol{d}\rangle$. If we fill it with the process $P = \overline{x}\langle \boldsymbol{y}\rangle$, we get $\mathcal{C}[P] = \left(c(\boldsymbol{x}).\overline{x}\langle \boldsymbol{y}\rangle\right) | \, \overline{c}\langle \boldsymbol{d}\rangle$. With this, we have a reaction $P \to P'$ of c receiving d, resulting in the substitution $\{d/x\}$, which gives us $P' = \overline{d}\langle \boldsymbol{y}\rangle \,|\, \mathbf{0}$. If we otherwise simply fill the context with $\mathbf{0}$, we would get $\mathcal{C}[\mathbf{0}] = c(\boldsymbol{x}).\mathbf{0} \,|\, \overline{c}\langle \boldsymbol{d}\rangle$, which results in a reaction $\mathcal{C}[\mathbf{0}] \to \mathbf{0} \,|\, \mathbf{0}$ with the same substitution as before. ◦

Definition 2.13 (Milner, 1999, p. 89)**.** Let $\cong$ be an equivalence relation over $\mathcal{P}$. We say that $\cong$ is a *process congruence* if it is preserved by all elementary contexts. This means that, if $P \cong Q$, then

$$\begin{gathered}\pi.P + R \cong \pi.Q + R \\ \nu x\, P \cong \nu x\, Q \\ P \mid R \cong Q \mid R \\ R \mid P \cong R \mid Q \\ !P \cong !Q.\end{gathered}$$

Definition 2.14 (Milner, 1999, p. 90)**.** Two process expressions P, Q are *structurally congruent*, written as $P \equiv Q$, if we can convert one into the other by using the following transformations — in whichever direction:

1. Change of bound names (*α-conversion*);
2. Reordering of terms in a summation;
3. $P \mid \mathbf{0} \equiv P,\ P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;
4. $\nu x\, \mathbf{0} \equiv \mathbf{0}$, $\nu x y\, P \equiv \nu y x\, P$,
   $\nu x\, (P \mid Q) \equiv P \mid \nu x\, Q$ if $x \notin \mathrm{fn}(P)$;
5. $!P \equiv P \mid !P$.

Example 2.10. Revisiting the processes P, P', P''' in Example 2.8, we can see that

a) $P \equiv d(\boldsymbol{z}).Q \mid \nu c\, \big(c(\boldsymbol{x}).\overline{x}\langle \boldsymbol{y} \rangle \mid \overline{c}\langle \boldsymbol{d} \rangle\big)$;

b) $P' \equiv \overline{d}\langle \boldsymbol{y} \rangle \mid d(\boldsymbol{x}).Q$;

c) $P'' \equiv Q\{y/z\}$.

In item a, this follows from transformation 3 in Definition 2.14. For items b and c, we can apply transformations 3 and 4 to eliminate both the restriction and the nil process, and in item b, we also applied transformation 1 to change the bound name z into x. ∘

Now that we have structural congruence, we can get processes to a *standard form* by applying the transformations iteratively until we can perform them no more.

Definition 2.15 (Standard form, Milner, 1999, p. 91)**.** A process expression is said to be in *standard form* if it has the form

$$\nu \vec{x}\, (Q_1 \mid \cdots \mid Q_m \mid !R_1 \mid \cdots \mid !R_n)\,,$$

where, for an index set I and $i, j \in I$:

- each Q_i is a non-empty sum $\sum_{k \in K} \pi_k.P_k$;
- each R_j is itself in standard form;
- in the case of $m = n = 0$, the process takes the shape $\nu \vec{x}\, \mathbf{0}$;
- if $\vec{x}$ is empty, then the restriction is omitted.

We finally have all we need to formally define reactions.

Definition 2.16 (Reaction, Milner, 1999, p. 91)**.** The relation $\rightarrow$ over the set of process expressions $\mathcal{P}$ defined by the rules in Table 5 is called the *reaction relation*:

Table 5
Rules of reaction

Rule	Reaction
Tau	$\tau.P + Q \rightarrow P$
React	$(c(\boldsymbol{x}).P + M) \mid (\overline{c}\langle \boldsymbol{y} \rangle.Q + N) \rightarrow \{y/x\}P \mid Q$
Par	$\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$
Res	$\dfrac{P \rightarrow P'}{\nu x\, P \rightarrow \nu x\, P'}$
Struct	$\dfrac{P \rightarrow P'}{Q \rightarrow Q'}$ if $P \equiv Q$ and $P' \equiv Q'$

In Definition 2.16, the rule Tau states that a process that performs an internal action τ can be reduced to its continuation. The React rule, as seen in Example 2.8, can occur between two processes that communicate in parallel over a shared channel, in which case we substitute the name held by the channel for the name that was sent.

The rules Par and Res, express closure under composition: if a process can react on its own, then it can also react when in parallel with another process, or when scoped to a name restriction, respectively.

Finally, the Struct rule permits reaction up to structural congruence, meaning that if two processes are structurally congruent, and one of them reacts, then the other also reacts, and both resulting processes remain structurally congruent.

2.3.3 Session types

The π-calculus by itself provides untyped channels, where any process can communicate any name at any time. While this provides a lot of flexibility and expression, it results in unsafe programs. We can have mismatched arities or unexpected values, leading to deadlocks or runtime errors. To address this, we introduce *session* types, which describe the structure of communication that a channel must follow.

Definition 2.17. *Session types* S are defined by the grammar:

$$S ::= !\sigma.S \;\Big|\; ?\sigma.S \;\Big|\; S \mid S \;\Big|\; \mu X.S \;\Big|\; X \;\Big|\; \textbf{end}$$

Here, σ represents a base type, such as integers or booleans. The constructs are interpreted as follows:

- $!\sigma.S$ prescribes the sending of a value of type σ and continuing the session as S;
- $?\sigma.S$ indicates the reception of a value of type σ and continuing as S;
- $S \mid T$ denotes parallel composition of independent sessions;
- $\mu X.S$ allows recursive protocols, where X is a variable for calling the recursion;
- **end** signifies the end of a session.

Duality. Each session type has a *dual*, which represents the complementary behavior in a communication.

$$(!\sigma.S)^\perp = ?\sigma.S^\perp$$
$$(?\sigma.S)^\perp = !\sigma.S^\perp$$
$$(S \mid T)^\perp = S^\perp \mid T^\perp$$
$$\mathbf{end}^\perp = \mathbf{end}$$

This feature ensures that two endpoints of a communication adhere to compatible protocols, that is, if one end is sending a message, the other end must be ready to receive it, and vice versa.

Recursive protocols. Session types support recursion, allowing the definition of protocols that can repeat certain patterns of communication. This is particularly useful for modeling long-lived interactions, such as a server that continuously handles client requests.

We read $\mu X.S$ as "the type S, where X can be replaced by the entire type itself". Whenever we encounter $\mu X.S$, we unfold the recursion by replacing the variabale X with the entire type $\mu X.S$:

$$\mu X.S \equiv S\{\mu X.S/X\}$$

It is required for recursive types to be *guarded*, meaning that the recursion variable X must appear within a send or receive action. This prevents the definition of unproductive types, such as $\mu X.X$, which would not correspond to any meaningful communication pattern. Duality is involutive and preserved under recursion, that is, $(S^\perp)^\perp = S$ and $(\mu X.S)^\perp = \mu X.S^\perp$.

Recursive session types can be unfolded as needed during type checking and process execution. Each cycle corresponds to one unfolding step at runtime.

Example 2.11 (Stream)**.** The type $\texttt{Stream} = \mu X.!\texttt{int}.X$ describes an endpoint that repeatedly sends integers. Its dual $\texttt{Stream}^\perp = \mu X.?\texttt{int}.X$ receives integers forever. Any finite run corresponds to a finite number of unfoldings. ∘

Typing rules. To enforce session types in the π-calculus, we introduce a typing system, where we extend the judgments of the untyped π-calculus to include an environment Γ. For instance, this is how we can type a handshake between two processes executing in parallel:

$$\frac{\Gamma, c : !\sigma.S \vdash P \quad \Gamma, c : ?\sigma.S^\perp \vdash Q}{\Gamma \vdash \overline{c}\langle v \rangle.P \mid c(\boldsymbol{x}).Q}$$

We need v to be of type σ. This rule enforces that the continuations P and Q respect the session continuation types. In Table 6 we present the complete set of typing rules for session types (Vasconcelos, 2009, p. 163).

For values, we assume the typing system described before in Section 2.1.3, where we have base types such as `int` and `bool`.

In Send-T, we assume a separate value typing that ensures the payload v has type σ, that the channel c has type $!\sigma.S$ in the context, and we also ensure that the continuation P respects the session type S. For Res, we consider two endpoints of a session, c and d, with dual types, and we ensure that the process P respects both types in its context.

Example 2.12. Consider the type

$$S = !\texttt{int}.?\texttt{bool}.\mathbf{end},$$

and its dual

$$S^\perp = ?\texttt{int}.!\texttt{bool}.\mathbf{end}.$$

Table 6
Typing rules for session types

Rule	Typing
Nil-T	$\dfrac{}{\Gamma \vdash \mathbf{0}}$
Send-T	$\dfrac{\Gamma \vdash c : !\sigma.S \quad \Gamma' \vdash v : \sigma \quad \Gamma'', c : S \vdash P}{\Gamma, \Gamma', \Gamma'' \vdash \overline{c}\langle v \rangle.P}$
Receive-T	$\dfrac{\Gamma \vdash c : ?\sigma.S \quad \Gamma', x : \sigma, c : S \vdash P}{\Gamma, \Gamma' \vdash c(\boldsymbol{x}).P}$
Par-T	$\dfrac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1, \Gamma_2 \vdash P \mid Q}$
Res-T	$\dfrac{\Gamma, c : S, d : S^{\perp} \vdash P}{\Gamma \vdash \nu cd\, P}$

A pair of processes typed with S and $S^{\perp}$, respectively, will first exchange an integer, and then a boolean, before terminating. An example of such processes would be:

$$P = \overline{c}\langle 42 \rangle.c(\boldsymbol{x}).\mathbf{0}$$
$$Q = c(\boldsymbol{x}).\overline{c}\langle \texttt{true} \rangle.\mathbf{0}$$

∘

Choice. Beyond simple send and receive actions, session types also support *choice* constructs, which allow a processes to have branches of execution where one side of the communication can *offer* multiple options, and the other side can *select* one of these. We extend the syntax of processes to include these constructs (Vasconcelos, 2009, p. 169):

$$\begin{array}{rll} P ::= & c \rhd \{\ell_i : P_i\}_{i \in I} & \text{offer} \\ & c \lhd \ell.P & \text{select } P \end{array}$$

When a process performs an offer on channel c, it provides multiple branches, each with a label ℓ_i and a corresponding process P_i. It will then wait until the other side selects one of these branches to continue. Conversely, when a process selects a branch on channel c, it just indicates which label ℓ_j it wants to proceed with, and then executes its own continuation P.

The corresponding session types are:

$$S ::= \&\{\ell_i : S_i\}_{i \in I} \;\Big|\; \oplus \{\ell_i : S_i\}_{i \in I}$$

Intuitively, the select and offer types are duals of each other, that is, $\oplus\{\ell_i : S_i\}_{i \in I}^{\perp} = \&\{\ell_i : S_i^{\perp}\}_{i \in I}$ and $\&\{\ell_i : S_i\}_{i \in I}^{\perp} = \oplus\{\ell_i : S_i^{\perp}\}_{i \in I}$.

Although the selecting process only chooses one branch, it must be prepared to handle any of the offered branches, as it cannot know in advance which one will be selected. For this reason, both the select and offer types include all possible branches. The typing rules are as follows:

$$\textsc{Sel-T}\quad \frac{\Gamma \vdash c\colon \oplus\,\{\ell_i\colon S_i\}_{i\in I} \quad \Gamma', c\colon S_j \vdash P \quad j \in I}{\Gamma, \Gamma' \vdash c \lhd \ell_j.P}$$

$$\textsc{Offer-T}\quad \frac{\Gamma \vdash c\colon \&\,\{\ell_i\colon S_i\}_{i\in I} \quad \Gamma, c\colon S_i \vdash P_i \quad \forall i \in I}{\Gamma, \Gamma' \vdash c \rhd \{\ell_i\colon P_i\}_{i\in I}}$$

Example 2.13 (Checkout)**.** With choice, we can model a checkout in a market, where the *client* selects a payment method or cancels the transaction, and the *server* offer these options, returning a boolean indicating whether the payment was successful or not. The session type for the client would be:

$$S = \oplus\begin{Bmatrix} \ell_{\mathsf{credit}} & : & !\mathrm{CreditCard}.!\mathtt{int}.?\mathtt{bool}.\mathbf{end}, \\ \ell_{\mathsf{debit}} & : & !\mathrm{DebitCard}.!\mathtt{int}.?\mathtt{bool}.\mathbf{end}, \\ \ell_{\mathsf{cancel}} & : & \mathbf{end} \end{Bmatrix}.$$

and the dual type for the server:

$$S^{\perp} = \&\begin{Bmatrix} \ell_{\mathsf{credit}} & : & ?\mathrm{CreditCard}.?\mathtt{int}.!\mathtt{bool}.\mathbf{end}, \\ \ell_{\mathsf{debit}} & : & ?\mathrm{DebitCard}.?\mathtt{int}.!\mathtt{bool}.\mathbf{end}, \\ \ell_{\mathsf{cancel}} & : & \mathbf{end} \end{Bmatrix}.$$

A client that chooses to pay the amount of 100 with a credit card, would be represented by

$$C = c \lhd \ell_{\mathsf{credit}}.\overline{c}\langle \boldsymbol{card}\rangle.\overline{c}\langle 100\rangle.c(\boldsymbol{x}).\mathbf{0},$$

where *card* is a value of type CreditCard. The server that can handle this request would be:

$$S = c \rhd \begin{Bmatrix} \ell_{\mathsf{credit}} & : & c(\boldsymbol{x}).c(\boldsymbol{y}).\overline{c}\langle \mathtt{true}\rangle.\mathbf{0}, \\ \ell_{\mathsf{debit}} & : & c(\boldsymbol{x}).c(\boldsymbol{y}).\overline{c}\langle \mathtt{true}\rangle.\mathbf{0}, \\ \ell_{\mathsf{cancel}} & : & \mathbf{0} \end{Bmatrix}.$$

If we compose the client and server in parallel, we get the following reaction sequence after the client selects the credit card option:

$$\begin{aligned} C \mid S &\to \overline{c}\langle \boldsymbol{card}\rangle.\overline{c}\langle 100\rangle.c(\boldsymbol{x}).\mathbf{0} \mid c(\boldsymbol{x}).c(\boldsymbol{y}).\overline{c}\langle \mathtt{true}\rangle.\mathbf{0} \\ &\to \overline{c}\langle 100\rangle.c(\boldsymbol{x}).\mathbf{0} \mid c(\boldsymbol{y}).\overline{c}\langle \mathtt{true}\rangle.\mathbf{0} \\ &\to c(\boldsymbol{x}).\mathbf{0} \mid \overline{c}\langle \mathtt{true}\rangle.\mathbf{0} \\ &\to \mathbf{0} \mid \mathbf{0} \end{aligned}$$

∘

Forwarding. Session types also allow for *forwarding*, also known as *linking*, which consists of connecting two channels such that communication on one endpoint is transparently relayed to the other. Forwarding is useful in scenarios where communication must be delegated between processes, allowing the interface of a service to be decoupled from its implementation and enabling more modular communication structures.

An example of forwarding is a server that receives requests on one channel and delegates the interaction to another process responsible for handling the session. Rather than explicitly mediating all communication between both channels, forwarding allows them to behave as directly connected endpoints.

We represent forwarding in the π-calculus with a special process called a *forwarder*, written as $c \leftrightarrow d$. Operationally, the forwarder behaves as an identity process, preserving the protocol structure between both channels by relaying messages bidirectionally.

The typing rule for forwarding is given by:

$$\frac{}{\Gamma, c{:}S, d{:}S^{\perp} \vdash c \leftrightarrow d}$$

The channels must have dual session types, ensuring that the communication actions expected on one endpoint are compatible with those provided by the other. This guarantees that forwarding preserves session fidelity and protocol correctness.

Connections to linear logic and game semantics. The relationship between the session-typed π-calculus and linear logic is well-established, with session-types corresponding to propositions in linear logic and processes corresponding to proofs (Wadler, 2012). This correspondence, known as the *propositions as sessions* paradigm, provides a deep theoretical foundation for understanding the behavior of concurrent processes and the structure of communication protocols.

Another strong connection to be made to the session-typed π-calculus is with games and strategies. In this setting, processes can be seen as strategies in a game where players (processes) interact according to the rules defined by session types. This perspective allows us to analyze the behavior of concurrent processes using game-theoretic concepts, such as winning strategies and equilibria, providing insights into the design and analysis of communication protocols in concurrent systems. We will further explore these connections in Chapter 3.

2.4 Rust

The implementation of YuugiLang is written in the Rust programming language. Rust is a is a modern systems program language that aims to provide both high performance and memory safety. It is also focused on concurrency, providing abstractions that make it easier to write safe concurrent code. In this section, we provide a brief overview of Rust's key features to motivate its use for this project.

2.4.1 Ownership and borrowing

One of Rust's most distinctive features is its ownership model, which is designed to ensure memory safety without a garbage collector. Every value has a single owner, and when the owner goes out of scope, the value is automatically deallocated (Steve Klabnik & Carol Nichols, 2019). This model prevents common memory errors such as dangling pointers and double frees, while also avoiding the overhead of garbage collection.

Example 2.14. In the following code snippet, a string is created and owned by the variable *s*.

```
1    {
2      let s = String::from("hello"); // s is the owner of the string
3
4      // do something with s
5    }// s goes out of scope and the string is deallocated
```

While inside the scope, we can use the string *s* freely. However, once we exit the scope, *s* is no longer valid, and the memory it occupied is automatically freed. ∘

There are two types of memory, stack and heap. Stack memory is used for storing values with a known, fixed size at compile time, such as integers and booleans, while heap memory is used for

values with a dynamic size, such as strings and vectors. Rust's ownership model applies to both stack and heap memory, ensuring that values are properly managed regardless of where they are stored.

Borrowing allows multiple references to a value without transferring ownership. There are two types of borrowing: immutable and mutable. Immutable borrowing allows multiple references to a value, but none of them can modify the value. Mutable borrowing only permits a single reference that can modify the value. This ensures that data is not modified while it is being read. For a more detailed explanation of ownership and borrowing, see Klabnik and Nichols (2019, Section 4).

2.4.2 Expressivity

The type system is very expressive, allowing for the definition of complex data structures and abstractions. It has suport for polymorphism, which allows for writing generic code that can work with different types; pattern matching provides a powerful way to deconstruct and analyze data; and traits enable defining shared behavior across different types. One notable feature is the `enum`, as it allows for defining types that can take on multiple forms, similar to algebraic data types in functional programming languages. This is particularly useful for modeling data that can have different variants, and is crucial for implementing the abstract syntax tree of YuugiLang. Paired with pattern matching, it provides a concise and expressive way to handle different cases based on the variant of the enum.

2.4.3 Error handling

A strong emphasis is placed on error handling, with a focus on distinguishing between recoverable and unrecoverable errors. Recoverable errors are handled using the `Result` type, which is an enum that can represent either a success (`Ok`) or an error (`Err`). This encourages developers to explicitly handle errors. Unrecoverable errors, on the other hand, are handled using the `panic!` macro, which causes the program to terminate. This distinction helps in writing robust code that can gracefully handle errors when possible. The `Option` type is also used to represent values that may or may not be present, further enhancing safety by avoiding null pointer dereferencing, as Rust does not have null values.

There is also a great incentive to create your own types for error handling, as the trait `Error` can be implemented for custom error types, and we use this feature extensively in the implementation of YuugiLang in order to provide meaningful error messages.

2.4.4 Concurrency

Ownership and borrowing rules extend to concurrent programming, making it easier to write safe concurrent code. This ensures that data shared between threads is either imutable or protected by synchronization primitives. As we use session types to model communication between concurrent processes in YuugiLang, this feature is particularly relevant. Data that needs to be shared between threads uses smart pointers, such as `Arc` (Atomic Reference Counted), which allows multiple threads to own a value, and `Mutex` (Mutual Exclusion), that provides safe access to shared data by guaranteeing that only one thread can access the data at a time. These are some of the reasons why Rust concurrency model is called "fearless concurrency" (Klabnik & Nichols, 2019, Section 16).

2.4.5 Ecosystem

Rust has a vibrant ecosystem with a rich set of libraries and tools. The package manager, Cargo, simplifies dependency management and project setup. The Rust community is active and welcoming, with a strong emphasis on documentation and best practices. This makes it easy to find resources and

support when working with Rust. The Rust Book (Klabnik & Nichols, 2019) is an excellent resource for learning Rust, and its documentation is comprehensive and well-maintained.

Overall, Rust's combination of performance, safety, and expressivity makes it an excellent choice for implementing a language like YuugiLang, which requires careful management of resources and concurrent processes.

Page left blank intentionally.

Chapter 3
Game Semantics

Game semantics is a logic formalism to express the behaviour of programs as an interaction between player and opponent. The Curry Howard isomorphism (Sørensen & Urzyczyn, 2006), creates a correspondence between logic and programming languages, where types relate to propositions and programs relate to proofs. We can make a similar relation in game semantics, where we can interpret games and strategies as *event structures (ESs)* (Castellan et al., 2017). In this chapter we will give a short introduction to main concepts of game semantics, that will later be used as the basis for the implementation.

In this chapter, we will follow the presentation of concurrent games based on event structures from Castellan et al. (2017) and show how session types and processes can be interpreted as arenas and strategies, respectively, following Castellan et al. (2020). We first introduce event structures and configurations, then we enrich then with polarity to define arenas. Then, we define strategies as particular event structures labelled by arenas, and finally we show how session types and processes can be interpreted as arenas and strategies.

3.1 Event Structures

When describing the evolution of a process, we can identify the observable actions as events, that is, if we have a soda machine that has a button to dispense a soda, we can define as events, for example, the insertion of a coin, the pressing of the button and the dispensing of the soda. The process of getting a soda from this machine can be described by

$$\text{coin.button.}\overline{\text{soda}},$$

where coin and button are input actions (or events) and $\overline{\text{soda}}$ is an output action. The behaviour of the machine in terms of input and output can be described by the following set:

$$SM = \{\epsilon, \text{coin}, \text{button}, \text{coin.button.}\overline{\text{soda}}\}.$$

In this set, we have ϵ representing the empty (or nil) action, the coin and button events, which can happen in isolation, and the sequence of events that actually gets a soda, which is coin.button.$\overline{\text{soda}}$. The last one gives us the idea of a *causal* relation between the events, which means that one event causes another event to happen. In the case of the soda machine, we can say that the insertion of a coin, succeeded by pressing of the button, causes the dispensing of the soda, but not the other way around.

Example 3.1 (Adapted from Castellan et al., 2017, pp. 4–6)**.** We can represent this relation as a directed graph, as in Figure 4, where the events are the nodes and the edges are the causal relations between them, that is, if a node has an edge to another node, it means that the first event causes the second event, and if more than one event has an edge to another event e, it means these events need to happen

before e can happen. For that reason, we say that the causality in ES is *conjunctive*, in opposition to *disjunctive*, because for an event to occur, all of its dependencies need to happen first.

In this example, the events coin and button are independent, as they can happen in any order, in which case we say they are *concurrent*. This means that the machine can accept the coin and the button press in any order, but both need to happen before the soda can be dispensed.

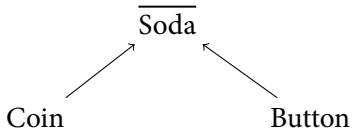

Figure 4
Representation of the soda machine events

∘

Definition 3.1 (Event structure, Castellan et al., 2017, p. 5)**.** An *event structure (ES)* is a tuple $(E, \preceq_E, \mathcal{C}_E)$, where E is a set of events, $\preceq_E$ is a partial order of events, which we call *causality*, and $\mathcal{C}_E$ is a non-empty set of finite subsets of E called *consistency*, such that

i) $\forall e \in E, [e] = \{e' \in E \mid e' \preceq_E e\}$, is finite;

ii) $\forall e \in E, \{e\} \in \mathcal{C}_E$;

iii) $\forall X \in \mathcal{C}_E, \forall Y \subseteq X, Y \in \mathcal{C}_E$;

iv) $\forall X \in \mathcal{C}_E, \forall e \in X, \forall e' \preceq_E e, X \cup \{e'\} \in \mathcal{C}_E$

Item *i* states that each event has finite number of causal predecessors. This can be understood as every event having a finite history of events that enabled it. In item *ii*, if an event is part of an ES E, it is consistent with itself. Consistency is down-closed, that is, if a set X of events is consistent, then any subset Y of X is also consistent, as in item *iii*. And finally, item *iv* tells us that if a consistent set $\mathcal{C}_E$ contains an event e, then it also contains all causal predecessors of e.

Definition 3.2. We say that a set $X \subseteq E$ of events in an ES E is *consistent* if $X \in \mathcal{C}_E$.

The states of an ES E are represented by *configurations*, and are the sets $X \subseteq E$ that are both consistent and *down-closed* . We write $\hat{\mathcal{C}}(E)$ for the set of configurations of E.

Definition 3.3. Given an ES $E = (E, \preceq_E, \mathcal{C}_E)$, we say that two events $e, e' \in E$ are in *conflict*, written $e \# e'$, if there is no configuration that contains both e and e'.

In other words, we can understand conflict as the complementary of consistency: a set of events is consistent when it has no conflicting pair of events.

A conflict relation # is *minimal*, written $\rightsquigarrow$, if for any $e, e' \in E$ such that $e \# e'$, there is no event $e'' \neq e'$ with $e \# e''$, and no event $e'' \neq e$ with $e'' \# e'$. That is, they are the only conflicting pair in $[e] \cup [e']$.

Example 3.2 (Adapted from Castellan et al., 2017, p. 6)**.** To exemplify configurations, the ES in Figure 5 shows a soda machine where when a coin is inserted, we get nondeterministally either a Cola or a Guaraná. The configurations of this ES are $\{\{\emptyset\}, \{\text{coin}\}, \{\text{coin}, \overline{\text{cola}}\}, \{\text{coin}, \overline{\text{guaraná}}\}\}$, and we cannot get both sodas at the same time, so we have a conflict.

∘

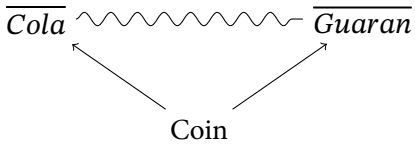

Figure 5
Nondeterministic soda machine

We define *immediate causality*, denoted by $e \to e'$, as the relation where $e \preceq_E e'$, with $e \neq e'$, and there is no e'' such that $e \preceq_E e'' \preceq_E e'$. In diagrams, we will only draw edges representing immediate causality or minimal conflicts, such as in Figure 5.

Example 3.3 (Adapted from Castellan et al., 2017, p. 6)**.** In Figure 6, we have a machine with two flavors of soda, *Cola* and *Guaraná*, and different buttons to dispense each flavor. The events work similarly to the previous examples, but as we have two conflicting options, we can get only one of them per coin. If both buttons were pressed after inserting a coin, we would have a non-deterministic choice.

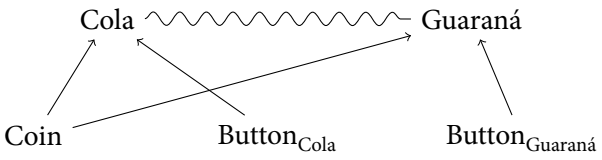

Figure 6
Representation of the soda machine events with conflicts

∘

Another way to enrich the concept of ES is to add the possibility for events to have polarity, that is, to be either input or output events. This is also what allows us to describe them as games with a Player and an Opponent.

Definition 3.4. An *arena* is an ES $(E, \leq, \mathcal{C})$ equipped with a polarity mapping $p : e \to \{\mathcal{P}, \mathcal{O}\}$ (Player/Opponent). The *dual* arena $A^{\perp}$ keeps the same events, causality and consistency, but reverses the polarity. We call events in A *moves*.

3.2 Strategies

A strategy describes the behaviour of a Player interacting with an Opponent inside an arena. Intuitively, a strategy determines which Player moves may occur in response to the moves performed by the Opponent.

Formally, a strategy is an event structure labelled by an arena, where the labelling specifies which moves of the arena are represented by the events of the strategy.

Definition 3.5 (Strategy)**.** A *strategy* on an arena A is an event structure with polarity S, together with a map

$$\sigma : S \to A$$

satisfying the following properties:

Consistency preservation If $\hat{C} \in \mathcal{C}(S)$, then $\sigma(\hat{C}) \in \mathcal{C}(A)$.

Local injectivity For all $s, s' \in \hat{C} \in \mathcal{C}(S)$, if $\sigma(s) = \sigma(s')$, then $s = s'$.

Polarity preservation For all $s \in S$,

$$p_A(\sigma(s)) = p_S(s).$$

Receptivity For every configuration $\hat{C} \in \mathcal{C}(S)$, if an Opponent move $s' \in A$ is enabled in $\sigma(\hat{C})$, then there exists a unique $s \in S$ such that $\sigma(s) = s'$.

Determinism If $\hat{C} \in \mathcal{C}(S)$ and s, s' are Player moves enabled at $\hat{C}$, then

$$\sigma(s) = \sigma(s') \implies s = s'.$$

Courtesy If $s \to s'$ and $p(s) = \mathcal{P}$, then $p(s') = \mathcal{O}$.

Consistency preservation means that if a set of events is consistent in the strategy, then the corresponding moves in the arena are also consistent. Local injectivity ensures that the strategy does not identify different events with the same move in the arena, which would make it impossible to distinguish between distinct occurrences of that move.

Polarity preservation states that the strategy preserves the polarity of moves, that is, if an event in the strategy is a Player move, then the corresponding move in the arena is also a Player move, and similarly for Opponent moves.

Receptivity ensures that whenever an Opponent move is enabled in the current configuration of the arena, the strategy is prepared to accept it. Moreover, there is a unique event in the strategy corresponding to that Opponent move.

Determinism indicates that if two Player events are enabled at the same configuration, then they must correspond to different moves in the arena. This guarantees that the strategy has no ambiguity in its responses to Opponent actions.

Finally, courtesy ensures that causal dependencies respect the interaction between Player and Opponent. In particular, if a Player move causally enables another move, then the enabled move must be an Opponent move. Thus, the strategy cannot internally generate chains of Player moves without interaction from the Opponent.

Example 3.4. Consider an arena where the Opponent may ask for a soda by performing the move coin, after which the Player may respond with either cola or guaraná.

A strategy for this arena may choose always to dispense Cola. In this case, whenever the Opponent performs the move coin, the strategy responds with the Player move cola, while never enabling the move guaraná.

This strategy is deterministic, since the Player has a unique response to each Opponent move, and receptive, since it always accepts the Opponent move coin. ∘

3.2.1 Interaction of Strategies

Given two strategies

$$\sigma : S \to A \qquad \text{and} \qquad \tau : T \to A^{\perp},$$

their interaction describes the synchronization of matching Player and Opponent moves.

Since the polarities of A and $A^{\perp}$ are opposite, events with matching labels may interact. The interaction combines the causal structures of both strategies into a joint event structure, written

$$\sigma \wedge \tau : S \wedge T \to A,$$

where compatible events are synchronized.

Example 3.5. Suppose one strategy expects the Opponent to perform the move coin, while another strategy produces the corresponding Player move coin. During interaction, these matching moves synchronize and become internal to the composed strategy.

Operationally, this resembles communication in the session π-calculus, where an output action synchronizes with a matching input action. ∘

A notable example of a strategy is the *copycat strategy*. Intuitively, the copycat strategy behaves as an identity process between two dual arenas by reproducing moves from one side onto the other. Whenever the Opponent performs a move in one arena, the copycat strategy responds by performing the corresponding move in the opposite arena, preserving the causal structure of the interaction.

Example 3.6. Consider the forwarding process

$$x \leftrightarrow y,$$

which connects two session channels x and y. Whenever a message is received on one channel, it is immediately forwarded to the other.

Under the game-semantic interpretation, this process corresponds to the copycat strategy, where moves performed on one side of the arena are reproduced on the opposite side. ∘

The copycat strategy plays a central role in game semantics, as it acts as the identity morphism under composition of strategies. Operationally, it resembles forwarding in the session π-calculus (discussed in Section 2.3.3), where communication received on one channel is transparently relayed to another.

3.3 From session types to arenas, and processes to strategies

We interpret session types as arenas and well-typed processes as strategies. Intuitively:

- a send/receive action corresponds to a question–answer pair on a move labelled by the payload type;
- selection/offer corresponds to a labelled branching in the arena;
- recursion in session types corresponds to recursive arena structure (guarded by polarity).

Formally, we give a compositional interpretation $[\![\circ]\!]$ on session types:

$$\begin{aligned}
[\![\mathbf{end}]\!] &= \text{the empty arena},\\
[\![X]\!] &= \text{the arena variable } X,\\
[\![!x.S]\!] &= \text{an arena where an Player move labelled } \mathbf{send}_\tau \text{ enables the continuation } [\![S]\!],\\
[\![?x.S]\!] &= \text{an arena where an Opponent move labelled } \mathbf{recv}_\tau \text{ enables the continuation } [\![S]\!],\\
[\![\oplus\{\ell_i : S_i\}_{i\in I}]\!] &= \mathcal{P} \text{ chooses one of the initial moves } \ell_i \text{ then continues with } [\![S_i]\!],\\
[\![\&\{\ell_i : S_i\}_{i\in I}]\!] &= \mathcal{O} \text{ chooses one of the } \ell_i \text{ then continues with } [\![S_i]\!],\\
[\![\mu X.S]\!] &= \text{the least fixpoint arena of the functor } X \mapsto [\![S]\!].
\end{aligned}$$

Duality of session types matches arena dual: $[\![S^\perp]\!] = [\![S]\!]^\perp$. The use of dual arenas reflects the interaction between a process and its environment: inputs expected by the process correspond to outputs provided by the environment, and conversely.

Example 3.7. Consider the session type

$$!\tau.?\sigma.\mathbf{end},$$

describing a protocol where a value of type τ is first sent, then a value of type σ is received, after which the session terminates.

Its associated arena consists of:

- a Player move labelled $\mathbf{send}_\tau$,
- followed by an Opponent move labelled $\mathbf{recv}_\sigma$,
- followed by termination.

∘

Typing ⇒ well-formedness of strategies. Session fidelity and linearity ensure that $[\![P]\!]$ satisfies receptivity (never blocks an $\mathcal{O}$-move offered by the environment) and determinism (no ambiguous $\mathcal{P}$ responses). Courtesy follows from the causal structure induced by the interpretation, where Player moves cannot internally justify other Player moves without an intervening Opponent move.

Restriction $\nu c\, P$ corresponds to the strategies interacting on the restricted channel followed by *hiding*, that is, removing the internal communication on the restricted channel from the external interface of the strategy. Parallel composition of processes corresponds to the interaction of strategies; selection/offer become the labelled branch/merge in $[\![\circ]\!]$.

A well-typed process judgment

$$\Gamma \vdash P : \Delta$$

is interpreted as a strategy

$$[\![P]\!] : \Big|\Big|_{c:S\in\Gamma} [\![S^\perp]\!] \longrightarrow \Big|\Big|_{c:S\in\Delta} [\![S]\!],$$

where the interface of the process is given by the parallel composition of the arenas associated with the session types appearing in the typing contexts.

Session fidelity and linearity ensure that $[\![P]\!]$ satisfies receptivity and determinism. Courtesy follows from the causal structure induced by the interpretation, where Player moves cannot causally justify other Player moves without an intervening Opponent move.

Restriction $\nu c\, P$ corresponds to composing strategies interacting on the restricted channel and then hiding the resulting internal communication from the external interface. Parallel composition of processes corresponds to the interaction of the associated arenas and strategies, while selection and offer correspond to labelled branching in the arena structure.

This interpretation provides a semantic view of session-typed processes as interactive behaviours, where communication corresponds to the exchange of moves between strategies and protocol correctness follows from the structure of the arena and the properties of strategies.

Chapter 4
Static Semantics

The syntax of the language is divided into two layers, the first is defined by Concurrent ML ($ML_{||}$) and the second is π_{DiLL} a session-typed π-calculus with *linear* session-types. The first layer is used for *data-level* computation, that is, expressions, references, functions, arithmentc operations etc. The second layer is used for *process-level* computation, that is, concurrency communication and channel structures. We can understand the first layer as a *payload*, which is the data that will be carried through the channels defined in the second layer. In this chapter we will explore the syntax of both layers, and in the next chapter we talk about the evaluation and execution of the language.

4.1 Abstract syntax of the payload layer

As previously discussed, the first layer of our language is $ML_{||}$, a simply-typed functional language with references. We assume the syntax and typing rules defined in Section 2.1.

We disguinish between *values*, which are terms that are fully evaluated, and *pure terms*, which are terms that can be evaluated further. We define them by using two `enum`, `Value` and `CmlTerm`.

Values. For values, we have the following grammar:

$$v ::= x \mid \underline{n} \mid \texttt{true} \mid \texttt{false} \mid \lambda x{:}\sigma.M \mid \ell \mid c \mid ()$$

This is adapted in the implementation as an `enum Value` with the variants:

- `Var`, represents a variable x, storing its name as a string;
- `Int`, is an integer literal, storing its value as an `i64`;
- `Bool`, a boolean literal, storing its value as a `bool`;
- `Unit`, the unit literal, storing no additional data;
- `Function`, represents a function abstraction, optionally storing the parameter name, its type, the body of the function, and an optional name for recursive functions;
- `Label`, is the label ℓ used in branching session types, storing its name as a string;
- `Channel`, the channel value, which wraps a `ChannelInternal`, that stores information about the channel, such as its name, queue of messages, protocol machine, etc. (detailed in Section 4.2)

In the concrete syntax of YuugiLang, most of these values are identified by their lexical structure. For example, integer literals are sequences of digits, boolean literals are given by the keywords `true` and `false`, and the unit literal is represented by the keyword `unit`. Variables are identifiers that do not match reserved keywords.

Function abstractions are introduced by the keyword `fun`, followed by a typed parameter, an explicit return type indicated by the arrow "→", and a body written between braces. Labels are identifiers used in branching constructs such as selection and offering. Channels are introduced dynamically through session constructs, typically using the `new` construct to create fresh endpoints.

Although not enforced by the parser, we recommend starting variable names with lowercase letters and channel names with uppercase letters to improve readability. The parser translates these surface forms into the corresponding abstract syntax tree representation used internally by the interpreter and type checker.

Table 7
Examples of concrete syntax and their corresponding AST representations

Concrete syntax	AST representation
`42`	`Value::Int`(42)
`true`	`Value::Bool`(`true`)
`unit`	`Value::Unit`
`x`	`Value::Var`("x")
`fun(x: int) → int { … }`	`Value::Function`(parameter = x, type = `int`, body = ⋯)
`Label(ok)`	`Value::Label`("ok")
`Channel`	`Value::Channel`(⋯)

Example 4.1. Consider the following function abstraction in YuugiLang:

```
fun(x: int) -> int {
  x + 1
}
```

This defines a function that takes an integer x and returns the result of adding 1 to it.

In the abstract syntax tree, this expression is represented as a `Function` value storing the parameter name x, its type `int`, and the function body. The body itself is represented by an `Add` node whose left operand is the variable x and whose right operand is the integer literal 1. ∘

Expressions. Expressions extend values with constructs that can be evaluated further. Their grammar is given by

$$M ::= v \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \mid M == M \mid M{+}M \mid MM \mid \texttt{ref}\ M \mid \texttt{get}\ M \mid \texttt{set}\ MM \mid \texttt{fix}\ x{:}\sigma M$$

In the concrete syntax of YuugiLang, these constructs use familiar programming-language notation. Conditional expressions are written using the keywords `if`, `then`, and `else`; arithmetic operations use infix notation; and function application is written by juxtaposition. References are manipulated through the keywords `ref`, `get`, and `set`. Recursive definitions are introduced with the keyword `fix`, while type annotations are written using the symbol `:`.

The parser translates these surface forms into the corresponding variants of the `CmlTerm` abstract syntax tree.

This grammar is adapted in the implementation as the *enum* `CmlTerm`, with the following variants:

- `Val`, which wraps a `Value`, representing a value;
- `If`, represents a conditional expression, storing the condition term, the `then` branch, and the `else` branch;
- `Equal`, compares two expressions for equality, storing the left and right operands;
- `Add`, represents integer addition, storing the left and right operands;
- `App`, represents function application, storing the function expression and the argument expression;
- `Ref`, creates a new reference, storing the expression whose value will be allocated;
- `Get`, dereferences a reference, storing the expression expected to evaluate to a reference;
- `Set`, updates the value stored in a reference, storing both the reference expression and the new value expression;
- `Fix`, defines a recursive function, storing the function name, its type, and the body of the λ-abstraction;
- `Annot`, represents a type annotation, storing the annotated expression and its type.

Concrete syntax	AST representation
`x + 1`	`CmlTerm::Add`(...)
`if x == 0 then { true } else { false }`	`CmlTerm::If`(...)
`f x`	`CmlTerm::App`(...)
`ref 42`	`CmlTerm::Ref`(...)
`get r`	`CmlTerm::Get`(...)
`set r 10`	`CmlTerm::Set`(...)
`fix fact: int → int { fun(x: int) → int { ... } }`	`CmlTerm::Fix`(...)

Table 8
Examples of concrete syntax and their corresponding expression AST representations

By structuring programs in this way, the AST makes explicit many conventions that are implicit in the surface syntax. Function applications are stored in a left-associative manner, so that a chain of applications is systematically represented as nested nodes. Conditionals always contain both branches, avoiding ambiguities such as dangling `else` constructs.

Arithmetic and comparison operators are represented by dedicated AST nodes, meaning that precedence and associativity are resolved during parsing and no longer need to be considered during

later compilation phases. The representation of references separates allocation, dereferencing, and updating into distinct forms, allowing the type checker to enforce the intended typing discipline.

Function abstractions and fixpoints carry explicit binding information, making variable scope explicit and avoiding accidental capture during substitution and evaluation.

This organization of the AST also provides the interface between the payload and process layers. Payload expressions such as integers, functions, labels, and channels may be communicated as messages in session interactions. By preserving type annotations, source spans, and binding information, the AST serves not only as the basis for type checking, but also as the foundation for evaluation and precise error reporting.

4.2 Abstract syntax of session layer

The second layer of YuugiLang is based on π_{DiLL}, a session-typed π-calculus with linear session types. While the payload layer is responsible for ordinary computation over values, the session layer describes concurrent processes and communication protocols.

The grammar of session processes is given by

$$
\begin{aligned}
P, Q ::= \; & \mathbf{0} \\
& \nu x\, P \\
& \overline{c}\langle \boldsymbol{M} \rangle.P \\
& c(\boldsymbol{x}).P \\
& c \lhd \ell.P \\
& c \rhd \{\ell_i : P_i\}_i \\
& P \mid Q \\
& !P \\
& X \\
& \mu X.P \\
& c \leftrightarrow d
\end{aligned}
$$

4.2.1 Session terms

As with the payload layer, session processes are represented in the implementation using an abstract syntax tree encoded as a Rust `enum`. Each constructor of the grammar corresponds to a variant of the enum representing session terms.

The main variants are:

- `New`, which creates a fresh channel and stores its name together with the continuation process;
- `Send`, representing $\overline{c}\langle v \rangle.P$, storing the channel, the payload value, and the continuation;
- `Receive`, representing $c(\boldsymbol{x}).P$, storing the channel, the bound variable, and the continuation;
- `Select`, representing labelled selection, storing the channel, the selected label, and the continuation;
- `Offer`, representing branching behaviour, storing a mapping from labels to continuation processes;
- `Parallel`, representing parallel composition of two processes;

- `Replicate`, representing a replicated process that may serve multiple sessions;
- `Rec`, representing recursive processes, storing the recursion variable and the process body;
- `Var`, representing recursion variables;
- `Forward`, representing forwarding (linking) between two channels;
- `End`, representing the terminated process.

In the concrete syntax of YuugiLang, session constructs are introduced through dedicated communication keywords. Sending is written using `send`, receiving with `recv`, labelled selection with `select`, and branching with `offer`. Parallel composition is written using the operator `|`, while recursion is introduced with the keyword `rec`. New channels are created using `new`, which binds a fresh channel name inside a process scope.

Example 4.2. The process

```
send c 42;
recv c as x;
end
```

first sends the integer 42 on channel c, then receives a value from the same channel and stores it in the variable x.

In the abstract syntax tree, this process is represented as a nested sequence of `Send` and `Receive` nodes, terminated by `End`. ∘

4.2.2 Channel internal

Channels are represented internally by the structure `ChannelInternal`, which stores the information required to associate runtime channels with their session protocols.

Conceptually, the structure connects three aspects of communication:

1. the identity of the channel;
2. the protocol currently associated with the channel;
3. metadata used to determine whether the channel behaves linearly or unrestrictedly.

The structure stores:

- the channel name;
- the protocol machine associated with the channel;
- a flag indicating whether the channel is linear;
- a flag indicating whether the channel has been linked (composed) with another channel;
- a reference to the linked channel when composition occurs.

The implementation provides several constructors for creating channels with different protocol configurations. The function `new` creates a fresh channel from a name, while `with_protocol` initializes a channel with a given session state. The constructors `restricted` and `restricted_with_action` create linear channels associated with restricted sessions.

Communication primitives such as `send`, `receive`, `handle_select`, `handle_offer`, and `link_channels` implement the protocol-level operations associated with session communication.

Finally, auxiliary functions such as `mark_composed` and `validate_final_state` are used to record channel composition and verify that a session terminates in a valid protocol state.

Additional runtime information associated with channels and their execution behaviour will be discussed in Chapter 5.

4.2.3 Session actions

The runtime representation of session protocols is given by the *enum* `SessionAction`. Each value of this enum corresponds to one stage of a session communication protocol.

The possible actions are:

- `Var`, representing recursive protocol variables;
- `Send`, indicating that a value of a given type must be sent;
- `Receive`, indicating that a value of a given type is expected;
- `Select`, representing labelled selection;
- `Branch`, representing labelled branching;
- `Rec`, representing recursive session behaviour;
- `End`, representing protocol termination.

Operationally, session actions act as runtime counterparts of session types. As communication progresses, channels transition between actions according to the structure of the protocol.

Several auxiliary operations are defined over session actions. The function `dual` computes the dual of an action, exchanging sends with receives and selections with branches. The predicate `is_dual_of` checks whether two actions are compatible communication partners.

Protocol progression is described by `next_state`, which computes the continuation of a session after a communication step. Recursive behaviour is handled through `unfold_recursive` and `can_unfold`, which implement guarded unfolding of recursive actions.

Additional helper functions such as `expected_type`, `allows_select`, and `allows_offer` provide validation of payload types and branching behaviour.

Together, session actions provide the connection between the static session type system and the operational behaviour of communicating processes.

4.2.4 Protocol machine

To ensure that runtime communication follows the session discipline, each channel is associated with a *protocol machine*. This component tracks the current session action and validates the evolution of the protocol as communication occurs.

The protocol machine stores:

- the current session action;
- a stack used for recursive unfolding;
- a history of previously executed protocol actions.

Whenever a communication step is performed, the protocol machine checks whether the requested operation is compatible with the current session state. If the transition is valid, the machine advances to the next protocol state.

The constructor `new` initializes a protocol machine from an initial session action. The function `validate_transition` checks whether a communication action is compatible with the current protocol state, while `process_state` computes the next protocol action and updates the machine accordingly.

The operations `transition` and `linked_transition` implement protocol evolution for ordinary and linked channels respectively.

Additional validation functions ensure protocol consistency. The predicate `validate_continuation` checks whether a continuation process matches the expected session state, while `validate_composition`

verifies that two protocol machines may be safely composed by checking that their current actions are dual.

Together, session actions and protocol machines provide the runtime representation of session protocols and ensure that communication remains consistent with the guarantees established by the static type system.

4.3 Type checking

Having fixed the abstract syntax for both payload expressions and processes, we now describe the type checking phase. Its role is to traverse the abstract syntax tree and decide whether a program is well-typed, that is, whether its payload expressions are consistent with the simply typed λ-calculus of Section 2.1 and whether its processes respect the discipline of session types introduced in Section 2.3.3. When a program is accepted, type checking also annotates each node of the AST with its type, so that later phases (such as evaluation and runtime execution) do not need to recompute typing information.

4.3.1 Environments

The implementation maintains two typing environments during type checking.

The first environment, written Γ, stores payload-level variables and their associated types. It is implemented as a stack of hash maps, allowing nested scopes to be introduced and removed during traversal of the abstract syntax tree. Entering the body of a function, conditional, or receive construct pushes a new scope onto the stack, while leaving the construct removes it.

The second environment stores session variables used in recursive session types. It is written Σ and is also implemented as a stack of hash maps. This environment is used to resolve recursive session variables introduced by recursive session definitions.

Formally, the checker maintains:

$$\Gamma : x \to \sigma \qquad \text{and} \qquad \Delta : X \to S.$$

The payload environment is manipulated through the operations `gamma_push`, `gamma_pop`, `gamma_insert`, and `gamma_lookup`, while the session environment uses the analogous operations `sigma_push`, `sigma_pop`, `sigma_insert`, and `sigma_lookup`.

Unlike the declarative system presented in Section 2.3.3, the implementation does not explicitly compute residual linear environments. Instead, linearity is enforced operationally through the protocol structure attached to channels and by checking that session actions follow the expected communication discipline.

4.3.2 Algorithmic type checking

The type checker is implemented as a recursive traversal of the abstract syntax tree. The implementation is divided into two mutually recursive functions:

```
cml : CmlTerm → Result<Type>
```

for payload expressions, and

```
session : SessionTerm → Result<Type>
```

for session processes.

Each function either returns the type of the term or produces a typing error explaining why the term is invalid.

Payload expressions. For payload expressions, the checker follows the structure of the simply-typed λ-calculus.

Variables are resolved through lookup in Γ. Integer, boolean, and unit literals return their corresponding base types directly.

For function abstractions, the checker requires an explicit parameter annotation. A new scope is introduced containing the parameter type, the body is checked recursively, and the resulting function type

$$\sigma_1 \to \sigma_2$$

is returned.

Function application first checks the type of the function expression. If the resulting type is a function

$$\sigma_1 \to \sigma_2,$$

then the argument expression must have type σ_1. Otherwise, the checker reports a function type mismatch.

Conditionals require the condition to have boolean type and both branches to return the same type. Arithmetic and equality operators require integer operands.

Reference operations are checked structurally:

- `Ref` constructs a reference type;
- `Get` requires a reference value;
- `Set` checks that the assigned value matches the reference contents.

Recursive functions are checked using the `Fix` constructor. The checker temporarily inserts the recursive function type into Γ before checking the body.

Session processes. Session processes are checked by inspecting the session structure attached to channels.

A channel value has type

$$c : S,$$

where S is represented internally as a `SessionAction`. Communication primitives verify that the current protocol action associated with the channel matches the operation being performed.

For a send process,

$$\bar{c}\langle \boldsymbol{v} \rangle.P,$$

the checker:

1. verifies that c has a session type beginning with a send action;
2. checks that the payload value has the expected type;
3. recursively checks the continuation P.

Receive processes are checked dually. The checker verifies that the channel expects a receive action, introduces the received variable into a fresh payload scope, and recursively checks the continuation.

Selection and offering inspect the branch structure stored in the session action:

- `Select` verifies that the selected label exists in the protocol;
- `Offer` verifies that all required labels are provided and recursively checks each branch.

Parallel composition recursively checks both subprocesses. Replication recursively checks the replicated process body.

Recursive session processes introduce a session variable into Σ, allowing recursive occurrences to be resolved through environment lookup.

Channel linking checks that the two channels carry dual session types. This is done using the helper function `is_dual_of`, which compares the corresponding `SessionAction` structures recursively.

The terminal process `End` always has type $\bullet$.

Example 4.3. Consider a simple client-server interaction in which one process sends an integer and another receives it. The session type associated with the sending endpoint is

$$S = !\texttt{int}.\mathbf{end},$$

while the receiving endpoint carries the dual type

$$S^{\perp} = ?\texttt{int}.\mathbf{end}.$$

The corresponding processes are

$$C = \overline{c}\langle 42\rangle.\mathbf{0} \qquad \text{and} \qquad S = \bar{c}(\boldsymbol{x}).\mathbf{0}.$$

When checking the client process C, the type checker first verifies that the channel c has a session type beginning with a send action. It then checks the payload expression 42, which has type `int`. Since this matches the type expected by the session action, the continuation $\mathbf{0}$ is checked recursively.

For the server process S, the checker verifies that the channel $\bar{c}$ expects a receive action carrying an integer. A fresh scope is introduced into the payload environment:

$$\Gamma, x : \texttt{int},$$

and the continuation $\mathbf{0}$ is checked under this extended environment.

Finally, when the channels are linked through restriction,

$$\nu c\, C \parallel S,$$

the checker verifies that the session types associated with c and $\bar{c}$ are dual. Since one endpoint sends an integer and the other receives it, the composition is accepted. ∘

Example 4.4. Consider a server that offers two options for payment, either by credit or by debit card, and a client that selects one of them. The session types are

$$A = \oplus\{\ell_{\mathsf{credit}} : !\mathsf{Card}.\mathsf{end},\ \ell_{\mathsf{debit}} : !\mathsf{Card}.\mathsf{end}\},$$

$$A^{\perp} = \&\{\ell_{\mathsf{credit}} : ?\mathsf{Card}.\mathsf{end},\ \ell_{\mathsf{debit}} : ?\mathsf{Card}.\mathsf{end}\}.$$

The client process

$$c \lhd \ell_{\mathsf{credit}}.\overline{c}\langle \mathit{card}\rangle.\mathbf{0}$$

is checked by verifying that the chosen label ℓ_{credit} is among the possibilities, and then ensuring that the payload *card* has type Card. The server process

$$\bar{c} \rhd \{\ell_{\mathsf{credit}} : \bar{c}(\boldsymbol{x}).\mathbf{0} \mid \ell_{\mathsf{debit}} : \bar{c}(\boldsymbol{y}).\mathbf{0}\}$$

is checked by verifying that all required branches are present and well-typed. Together, the processes form a safe interaction: whichever branch the client selects, the server has a matching continuation. ∘

4.3.3 Error reporting

When type checking fails, the system produces diagnostic messages anchored to the source span of the offending AST node. Typical errors include mismatches between expected and actual payload types at communication points, attempts to reuse a linear channel after it has been consumed, unoffered labels in an offer construct, or unconsumed channels left at the end of a process. By maintaining both Γ and Δ explicitly during traversal, the checker is able to give precise feedback to the programmer about the source and nature of each violation.

4.4 Parsing

The static semantics operates on the AST of the language, rather than on the raw source code. In this section we will explore the concrete syntax of YuugiLang, the grammar and the parsing process that transforms source code into an AST.

4.4.1 Concrete syntax vs. abstract syntax

While the programmer interacts with the language through its *concrete* syntax, we have to translate it to the *abstract* syntax that we defined in the previous sections. The AST is a tree representation of the structure of the source code, where each node represents a construct in the language. As previously discussed, the AST is divided in two layers:

- $\text{ML}_{||}$ terms (pure expressions or *payloads*): variables, abstractions, aplications, integers, booleans, etc.
- Session terms (processes): channel creation, send, receive, select, offer, etc.

Each AST node stores a span, which is a range of bytes in the source code that corresponds to the construct it represents. In the case of key nodes, like channel creation, send and receive, we also store a token that represents the construct.

4.4.2 Lexical structure

The lexical structure of YuugiLang is defined by the following elements:

Identifiers: names for variables and channels. They must start with ASCII letters or underscores, but can contain digits as well. Examples: `x`, `my_var`, `_channel1`.

Although not enforced by the parser, we recommend starting variable names with lowercase letters and channel names with uppercase letters to improve code readability.

Keywords: a few keywords are reserved for special constructs in the language (such as `if`, `then`, `else`, `fix`, `send`, `recv`, `select`, etc.) and cannot be used as identifiers. For the full list of keywords, see Table 9.

Literals: integer (e.g., `42`), boolean (`true`, `false`), and the unit literal `()`.

Punctuation: symbols used to structure the code, such as parentheses `( )`, braces `{ }`, commas `,` (used for argument/operand separator), semicolons `;` (independed sequencing), dots `.` (binded sequencing), pipes `|` (choice separation).

Operators: symbols used for operations, such as plus `+`, equal `==`, parallel `||`, etc.

Comments: single-line comments start with `//` and extend to the end of the line; multi-line comments are enclosed between `/*` and `*/` and can span multiple lines. It is not attached to any node in the AST and is ignored during parsing.

In Table 9 we present the complete grammar for YuugiLangin the extended Backus-Naur form (EBNF). We can observe in two main parts: one that concerns to $ML_{||}$, and one related session terms. The grammar is designed to be unambiguous and to respect operator precedence and associativity.

Table 9
Grammar for YuugiLang.

Id	::=	`[a-zA-Z_][a-zA-Z0-9_]`(*)
Value	::=	Id | *n* | `true` | `false` | `()` | `fun (`Id : Type`)` → Type `{` Expr `}` | Label | Channel
Type	::=	`int` | `bool` | `unit` | `ref<`Type`>` | Type → Type | `Chan<`Action`>`
Expr	::=	`if` Expr `then` Expr `else` Expr | Expr `==` Expr | Expr `+` Expr | Expr Expr | `ref` Expr | `get` Expr | `set` Expr = Expr | `fix (`Id : Type → Type`)` Expr | Expr : Type | Value | `(`Expr`)`
Session	::=	`new` Id : Action `;` Session | `send` Expr `,` Expr `;` Session | `recv` Expr `,` Id `;` Session | `select` Expr `,` Label `;` Session | `offer` Expr `{` Branch (`|` Branch)(*) `}` | Session `||` Session | `!`Session | `rec` RecVar `.`Session | `link (`Expr `,` Expr`)` `;` Session | `end` | RecVar | `(`Session`)`
Branch	::=	Label ⇒ Session
Action	::=	`send<`Type`>;` Action | `recv<`Type`>;` Action | `select {` Label : Action (`|` Label : Action)(*) `}` | `offer {` Label : Action (`|` Label : Action)(*) `}` | `rec` RecVar `.`Action | `end` | RecVar | `(`Action`)`
Label	::=	Id
RecVar	::=	Id

The grammar above describes the concrete syntax accepted by the parser. Internally, parsing produces the AST structures introduced in the previous section, where precedence and associativity are made explicit through the nesting of nodes.

4.4.3 Parsing strategy

We use a two-phase parsing strategy to transform source code into the AST. In the first phase, a *lexer* (or *tokenizer*) scans the source code and converts it into a sequence of tokens, which are the basic building blocks of the language (keywords, identifiers, literals, operators, etc.). In the second phase, a *parser* takes the sequence of tokens and constructs the AST

Pure layer. For infix operators, that is `+` and `==`, and left-associtive application we use the *precedence-climbing* algorithm. This checks the tokens created by the lexer and builds the AST respecting the precedence and associativity of the operators. The other constructs are parsed using a recursive descent approach, where each non-terminal in the grammar corresponds to a function in the parser.

The precedence levels we use are:

1. Function application, left associative (highest precedence).
2. Unary operators: `ref`, `get`.
3. Addition (`+`), left associative.
4. Equality (`==`), non-associative.
5. Conditional (`if` … `then` … `else`), right associative (lowest precedence).

To respect this precedence, when we call the `parse_cml_term` function, we pass the lexer to the minimium precedence, that is, we go through a function called `parse_cml_if`, which checks if the next token is `if`, and if so, it parses the condition, then the `then` branch, and finally the `else` branch. If the

next token is not `if`, it calls `parse_cml_eq`, which does the same step for the equality operator, next we go to `parse_cml_add`, which checks for addition, and finally we reach `parse_cml_app_or_annotation`, which handles function application and type annotations. This function calls `parse_cml_atom`, which handles the atomic constructs, such as literals, variables, abstractions, parentheses, fixpoints, `ref`, `get`, `set` operation.

After getting an atom, we check if we have a colon, indicating a type annotation, which if we do, we parse the type and return the annotated term. If there is no annotation, we loop through the tokens, breaking if the next token is not an atom (there is no token), and if it is, we parse the next atom, create an application node with the current term and the new atom, and continue the loop.

The atoms are parsed using recursive-descent, where each construct matches a pattern of tokens. This means that we recursively traverse the lexer tokens and match them against the expected patterns .For example, if the next token is `fun`, we expect left parenthesis, then an identifier, a colon, a type, a right parenthesis, an arrow, a left brace, a new term, and finaly a right brace, creating an abstraction node with the parsed components.

Session layer. The session layer is parsed using a recursive-descent approach as well. We call `parse_session_term`, which passes the lexer to `parse_session_parallel`. This function checks for a replication, which can be either a `!` followed by another replication or a primary session term. After getting a replication, we check if the next token is `||`, indicating a parallel composition, and if so, we parse the next replication and create a parallel node with the current term and the new replication, continuing the loop until there are no more `||` tokens. What this shows is that the session layer folows the following precedence:

- `()` groups.
- Replication (`!`), which binds the most tightly.
- Parallel composition (`||`).

As the continuation of session terms is always explicit, we do not need to handle precedence for the other constructs, which are all parsed in `parse_session_primary`.

4.4.4 Error handling

During parsing, we also store a cursor that indicates the current position in the source code. Whenever the parser encounters an unexpected token or an invalid construct, it generates a descriptive error message that includes the the file, line and column number where the error occurred. This information is crucial for debugging, as it helps the programmer quickly locate and fix issues in their code.

The error handling mechanism is implemented using Rust's `Result` type, which is represented by a pair of types: `Ok(T)`, which indicates a successful operation returning a value of type `T`, and `Err(e)`, which indicates an error occurred, returning an error value of type `e`. We create a custom error type implementing the `Error` trait, `ParseError`, that encapsulates the error message and the span where the error occurred.

Chapter 5
Dynamic Semantics

In this chapter we will present the dynamic semantics of YuugiLang, which is composed by the evaluation process and the runtime system. The evaluation process is responsible for transforming a program into a result, while the runtime system is responsible for managing the execution of the program, including memory management and handling of side effects.

5.1 Evaluation

The evaluation process is divided into two main parts: expression evaluation and game evaluation. The first one is responsible for evaluating expressions of $ML_{||}$, while the second one is responsible for evaluating game constructs, which is done through the communication between channels.

5.1.1 Expression Evaluation

Expression evaluation is done by the recurive call of the function `eval_pure`, which takes an $ML_{||}$ term, an environment and a memory unit as input and returns `Result`, that can be either a `Value` or an error. We define the function through a pattern matching on the term for each term in the *`enum`* `CmlTerm` defined in Section 4.1.

Values. For this case, the function simply returns the value itself, since values are already in their evaluated form. The only exception is in the case of a variable, where we call the function `lookup_var` to find its value in the environment, returning an error if the variable is not found. In Listing 1 we show the code for the first three cases of value matching pattern.

Applications. For function applications, we first call `eval_pure` on both the function and the argument. If the function evaluates to an abstraction (`Value::Lambda`), we create a new scope by cloning the current environment and then inserting the value of the argument via `insert_at_new_scope`. Finally, we call `eval_pure` on the body of the abstraction with the new environment. The code for this case can be seen in Listing 2.

Fixpoint. Here, we define a new abstraction that uses the variable name and body of the fixpoint term. We then insert a variable on the environment with the name of the fixpoint term and the abstraction as a value. Finally, we evaluate the body of the fixpoint term with the new environment. This can be seen in Listing 3.

Conditionals. This case simply evaluates the conditional term, and if the value is a boolean, it evaluates the first branch if the value is true, or the second branch if the value is false.

Summing. We evaluate each side of the summing terms. If both sides evaluate to integers, we return the sum of both integers as a value.

```
        Val(v) => match v {
            Var(name) => {
                let var = lookup_var(env, name);
                if let Some(var) = var {
                    Ok(var)
                } else {
                    Err(Box::new(UndefinedVariable(name.clone())))
                }
            }
            Lambda {
                name: self_name,
                param,
                param_type,
                body,
            } => Ok(Lambda {
                name: self_name.clone(),
                param: param.clone(),
                param_type: param_type.clone(),
                body: body.clone(),
            }),
            Int(i) => Ok(Int(*i)),
            Bool(b) => Ok(Bool(*b)),
            Channel(internal) => Ok(Channel(internal.clone())),
            Unit => Ok(Unit),
            Address(addr) => Ok(Address(*addr)),
            Label(label) => Ok(Label(label.clone())),
        },
```

Listing 1
Evaluation of values.

```
        App(f, arg) => {
            let f_val = eval_pure(f, env, memory)?;
            let arg_val = eval_pure(arg, env, memory)?;
            if let Lambda { name, param, body, .. } = f_val {
                let mut new_env = env.clone();
                if let Some(self_name) = name {
                    let self_val = Lambda {
                        name: Some(self_name.clone()),
                        param: param.clone(),
                        param_type: None,
                        body: body.clone(),
                    };
                    insert_at_new_scope(&mut new_env, self_name, self_val);
                } else {
                    insert_at_new_scope(&mut new_env, param.clone(), arg_val.clone());
                }
                insert_var(&mut new_env, param.clone(), arg_val);
                eval_pure(&body, &new_env, memory)
            } else {
                Err(Box::new(ExpectedFunction))
            }
        }
```

Listing 2
Evaluation of applications.

```
77          Fix(f, _annot, body) ⇒ match body.as_ref() {
78              Val(Lambda{param, param_type, body, ..}) ⇒ Ok(Lambda {
79                  name: Some(f.clone()),
80                  param: param.clone(),
81                  param_type:  param_type.clone(),
82                  body: body.clone(),
83              }),
84              _ ⇒ Err(Box::new(ExpectedRecursiveLambda)),
85          },
```

Listing 3
Evaluation of fixpoints.

Equality. Similar to summing, we evaluate both sides of the equality term. We then simply do the comparison using the `==` operator from Rust, then returns the result as a boolean value.

Memory. For memory operations, we have three cases: allocation, retrieval and update. For allocation, we evaluate the term to be stored in memory, then we call the function `alloc`, and return the address as a value. For retrieval, we call the evaluation, if it corresponds to an address, we call the function `get` to return the value stored at that address. For update, we first evaluate the term that corresponds to the address, if it is in fact an address, we then evaluate the term to be stored in memory, call `set` to update the value at that address, and return this value. We show the code for these cases in Listing 4.

```
111         Ref(t) ⇒ {
112             let t_val = eval_pure(t, env, memory)?;
113             let addr = memory.alloc(t_val);
114             Ok(Address(addr))
115         }
116
117         Get(t) ⇒ {
118             let addr_val = eval_pure(t, env, memory)?;
119             if let Address(addr) = addr_val {
120                 let value = memory.get(addr)?.clone();
121                 Ok(value)
122             } else {
123                 Err(Box::new(ExpectedReference))
124             }
125         }
126
127         Set(t1, t2) ⇒ {
128             let addr_val = eval_pure(t1, env, memory)?;
129             if let Address(addr) = addr_val {
130                 let value = eval_pure(t2, env, memory)?;
131                 memory.set(addr, value.clone())?;
132                 Ok(value)
133             } else {
134                 Err(Box::new(ExpectedReference))
135             }
136         }
```

Listing 4
Evaluation of memory operations.

5.1.2 Game Evaluation

The evaluation of a game corresponds to the creation of an *arena*, which is a directed acyclic graph (DAG) that represents the flow of the game. Each node in the arena represents a state of the game,

while the edges are the possible transitions (the *moves* or *events*) between states. The arena is built through communication between channels, where each represents a player (either a *proponent* or an *oponent*) in the game.

We define the `enum Player` to represent the two types of players in the game: `Proponent` and `Opponent`. Another important type is the `enum Event`, which represents the possible moves that can be made in the game. These events correspond to the session terms, and can be: the creating of a new channel, sending a value through a channel, receiving a value from a channel, or offering a choice in a channel, selecting a choice in a channel, linking two channels, unfolding a recursive session, or terminating a session.

Next, we define the `struct DAGNode`, which represents a node in the arena. Each node contains an identifier, the event that led to it, the player who made the move, the justification (the previous node), and a list of child nodes (the possible next states of the game). The arena itself is defined by the `struct Arena`, which contains two vectors: one of nodes, and one of the edges (a pair of integers identifying the parent and the child).

We need a few helper functions to manage the arena. The function `add_node` adds a new node to the arena, this is done by creating a new `DAGNode` and pushing it to the vector of nodes in the arena. If this node has a justification (a parent node), we also add an edge pair to the vector of edges in the arena. The function `merge` merges two arenas by appending the nodes and edges of the second arena to the first one, adjusting the identifiers of the nodes in the second arena to avoid conflicts. The function `validate_justification` serves to check if a node's justification is valid, i.e., if the justification node exists in the arena and if the player making the move is different from the player who made the justification move. We also have the function `print_arena` to visualize the arena structure, which is useful for debugging purposes and for understanding the flow of the game.

The last constructor we need for evaluating games is the `struct Process`. Each process is formed by a session term, an environment, a memory, the node identifier where the process is currently at, and the causal context, which is a vector of node identifiers representing the history of moves leading to the current state.

We can now define the function `eval_game`, which takes a process and an arena as input, and returns a result, either a pair containing a value (which is always `Unit` for game evaluation) and the updated arena, or an error. With `eval_pure` we define this function through a pattern that recurively evaluates each session term defined in Section 4.2.

New Channel. For this case, we create a new node in the arena with the event `NewChannel`, the player as `Proponent`, the justification as the current node of the process, and no children. Then, we create a restricted channel internal, using the new name and the node identifier of the newly created node. We also insert the new channel in the environment of the process. Finally, we update the process to have the new node identifier and the term as the continuation of the new channel, and call `eval_game` to continue the evaluation. The code for this case can be seen in Listing 5.

Send. Here, we first call `eval_pure` to evaluate both the name and the message to be sent. If the name evaluates to a channel, we add a new node from the proponent perspective, copying the last causal context node of the current process as justification. We call the `send` function of the channel internal to send the message value, update the `last_send_node` entry of the internal, and update the process to have the new node identifier and the term as the continuation of the send operation. At last, we call `eval_game` again to continue the process. We show the code for this case in Listing 6.

Receive. This case is similar to send, but from the opponent perspective. We evaluate the name, if it is a channel, we call the `receive` function of the channel internal to get the message value. We then add a new node from the opponent perspective, using the `last_send_node` of the internal as justification. We update the `last_receive_node` entry of the internal, and update the process to have the new node identifier, the term as the continuation of the receive operation, and insert the received message in the

```
        SessionTerm::NewChannel(name, opt_proto, body) => {
            let fresh_name = name.to_string();
            let parent = process.causal_context.last().copied();
            let node_id = add_node_sync(
                arena,
                Event::NewChannel(fresh_name.clone()),
                Player::Proponent,
                parent,
            );
            process.current_node = Some(node_id);

            let internal = match opt_proto.as_ref() {
                Some(s) => ChannelInternal::restricted_with_action(&fresh_name, s.clone(), node_id),
                None => ChannelInternal::restricted(&fresh_name, node_id),
            };
            let channel = Value::Channel(Arc::new(Mutex::new(internal)));

            let mut next = process.with_causal_node(node_id);
            insert_var(&mut next.env, name.clone(), channel.clone());
            next.term = *body;
            eval_game(&mut next, arena)
        }
```

Listing 5
Evaluation of new channel.

```
        SessionTerm::Send(channel_term, message_term, cont) => {
            let channel_val = eval_pure(&channel_term, &process.env, memory)?;
            let message_val = eval_pure(&message_term, &process.env, memory)?;

            if let Value::Channel(arc_channel) = channel_val {
                let ch_name = { arc_channel.lock().unwrap().name.clone() };
                let parent = process.causal_context.last().copied();
                let node_id = add_node_sync(arena, Event::Send(ch_name), Player::Proponent, parent);

                ChannelInternal::send(&arc_channel, message_val)?;

                {
                    let mut c = arc_channel.lock().unwrap();
                    if let Some(peer_w) = &c.linked_to {
                        if let Some(peer) = peer_w.upgrade() {
                            peer.lock().unwrap().last_send_node = Some(node_id);
                        } else {
                            c.last_send_node = Some(node_id);
                        }
                    } else {
                        c.last_send_node = Some(node_id);
                    }
                }

                let mut next = process.with_causal_node(node_id);
                next.current_node = Some(node_id);

                next.term = *cont;
                return eval_game(&mut next, arena);
            }

            Err(Box::new(ExpectedChannel))
        }
```

Listing 6
Evaluation of send.

environment. We end by calling `eval_game` on the continuation.

SELECT. For this term, we evaluate both the channel name and the label to be selected. If we have a channel, we add a new node from the proponent perspective, using the last causal context node of the current process as justification. The next step is to push the new node to the causal context, and call the `handle_select` function of the channel internal to perform the selection on the opponent side. As select also has a continuation term, the `handle_select` function updates the process term for it, so we just need to call `eval_game` again on the process.

OFFER. We create a vector of processes, that we will call `payload_branches`, one for each label in the offer term. We evaluate each label individually, and for each one we add a pair to the vector, containing the label value and the correspoding continuation body. After this we call the `handle_offer` function of the channel internal, which will wait for the proponent to make a selection, and then update the process term to the corresponding continuation. Finally, we call `eval_game` on this updated process. This is shown in Listing 7.

```
479         SessionTerm::Offer(channel_term, branches) => {
480             let channel_val = eval_pure(&channel_term, &process.env, memory)?;
481
482             if let Value::Channel(arc_channel) = channel_val {
483                 let ch_name = { arc_channel.lock().unwrap().name.clone() };
484                 let parent = process.anchor_parent();
485                 let node_id = add_node_sync(arena, Event::Offer(ch_name), Player::Opponent, parent);
486                 let mut next = process.with_causal_node(node_id);
487
488                 let mut payload_branches = Vec::with_capacity(branches.len());
489                 for (label, body) in branches {
490                     let memory = &mut next.memory;
491                     let label_val = eval_pure(&label, &next.env, memory)?;
492                     payload_branches.push((label_val, *body));
493                 }
494
495                 let _chosen =
496                     ChannelInternal::handle_offer(&arc_channel, payload_branches, &mut next)?;
497
498                 next.current_node = Some(node_id);
499                 return eval_game(&mut next, arena);
500             }
501
502             Err(Box::new(ExpectedChannel))
503         }
```

Listing 7
Evaluation of offer.

PARALLEL COMPOSITION. For this case, we create two new processes, one for each session term in the parallel composition. We then call `eval_game` on both processes, obtaining two arenas, which we merge using the `merge` function. At last, we add a new node with the event `Join`, executed by the proponent. The value returned is always `Unit`, as parallel composition does not produce a value.

REPLICATE. We create two processes with the same session term, evaluate both using `eval_game`, merge the resulting arenas, and return the merged arena along with result of the evaluation.

LINK. We evaluate both channel names, if they are channels, we add a new node with the event `Link`, executed by the proponent. We then call the `link_channels` function from the first channel internal, passing the second channel internal as an argument. The value returned is always `Unit`.

Recursion unfolding. We add a new node with the event `RecUnfold`, executed by the proponent. We then update the process to have the body of the recursive session as the new term, and call `eval_game` again.

Termination. We add a new node with the event `End`, executed by the proponent. The value returned is always `Unit`.

5.2 Runtime System

The runtime system of YuugiLang is responsible for the execution of the session programs, synchronizing endpoints, and producing the observable behavior of the program, represented by the arena.

5.2.1 Architecture

The runtime is executed in two layers:

- a *pure* evaluator for $\mathrm{ML}_{||}$ terms, with a per-process memory and environment;
- a *session* executor that manages the channel handshakes, and records the events in a shared arena.

Each process carries the current session term being evaluated, a memory and environment for the pure evaluation, causal pointers (`current_node`, a stack of `causal_context`, and an optional *parallel anchor*, used to for the parental relationship between cross-endpoint interactions).

5.2.2 Execution and concurrency

When executing the program, we check for parallel composition or replication session terms. In these cases, we run the function `fork_for_parallel` twice, which takes as argument a process and a parent node, and returns a new process. We then spawn two threads, one for each process, and evaluate both using `eval_game`. After both threads finish, we add a new node with the event `Join`, executed by the proponent, with the parent node as justification. The value returned is always `Unit`, as parallel composition and replication do not produce a value.

5.2.3 Shared Arena

The runtime system is composed by a shared arena, which is a wrapper around a `struct Arena` protected by an `Arc<Mutex>` to ensure safe concurrent access when executing multiple processes in parallel. The helper function `add_node_sync` locks the arena, safely adds the node, pushes the edge (parent → id) when it exists, and then unlocks it. This ensures thread-safe, linearizable updates to the arena.

5.2.4 Channels and handshakes

Each channel tracks the expected next action through its protocol machine, a link to its peer channel, and the causal pointers of the last send and receive actions. When a process performs a send, it stores the node identifier in the `last_send_node` field of the channel internal, which is used as justification for the next receive action on the peer channel. This ensures that the causal relationship between send and receive actions is maintained in the arena.

5.2.5 Pure memory management

Each process has its own memory, which is a simple key-value store mapping addresses to values. Whenever a process forks, it clones its memory, ensuring that each process has its own independent memory space. This is important to prevent side-effects interfering between processes, as each process can modify its own memory without affecting the others.

5.2.6 Validation of moves

To ensure the correctness of the arena, we have the function `validate_justification`, which checks if:

- every node's parent exists in the arena;
- the player making the move is different from the player who made the justification move.
- each *receive/select* event is justified by the corresponding *send/offer* event.

This function is called whenever a new node is added to the arena, and if any of these conditions are not met, an error is returned, indicating a violation of the game semantics.

5.2.7 Errors and diagnostics

The runtime system is designed to handle errors gracefully, providing meaningful diagnostics to help identify and fix issues in the session programs. Errors can be categorized into:

- *Parsing errors*, which occur when the input program does not conform to the syntax of Yuugi-Lang.
- *Typing errors*, which are detected during the parsing and type-checking phases before execution.
- *Memory errors*, such as accessing an uninitialized address or attempting to update a non-existent address.
- *Session errors*, such as attempting to perform an action that is not allowed by the current protocol state, or mismatched send/receive actions.
- *Arena errors*, which are detected by the `validate_justification` function, indicating violations of the game semantics.

Errors that occur during the execution of parallel/replicate terms are propagated at the join. For this reason, any function that returns a `Result` propagated at runtime must have the error type as `Box<dyn Error + Send + Sync>`. If we did not have to move errors through threads, having `Box<dyn Error>` would suffice, which propagates any error that implements the `Error` trait, but for sharing errors between threads, we need to ensure that the error also has `Send` and `Sync` traits.

When an error occurs, the runtime system captures the error and provides a detailed message, including the type of error, a description of the issue, and the location in the source code where the error occurred.

For debugging purposes, the runtime system includes a function to print the current state of the arena in a DOT format (graph description language), which can be useful for understanding the flow of the game and identifying any issues with the session interactions.

5.2.8 Program execution

The execution steps of a YuugiLang program are as follows:

1. The program is parsed to create an abstract syntax tree (AST).
2. The AST is type-checked to ensure that the program adheres to the type rules of YuugiLang.
3. If the program is well-typed, we execute the `run` function from the `Runtime` struct, which initializes the shared arena and creates the initial process with the main session term, an empty environment and memory, and the initial node identifier and causal context.
4. The `eval_game` function is called on the initial process, which evaluates the session term, managing the communication between processes through channels, and building the arena with the events that occur during the execution.
5. Once the evaluation is complete, the final arena is validated using the `validate_justification` function to ensure that it adheres to the game semantics.
6. If the arena is valid, it is returned as the result of the program execution, along with a `Unit` value, indicating successful completion.

Page left blank intentionally.

Chapter 6
Considerations

This last chapter reflects on the development of YuugiLang, considering its educational value, limitations, lessons learned during the implementation, and possible directions that can be explored in future work.

6.1 Educational value

The primary goal of this work was to create a minimal programming language that demonstrates the core concepts of game semantics in a practical setting. By developing a minimal interpreter, we were able to concretely illustrate how the abstract notions of strategies, duality and interaction can be built into a working language.

By showing how the theoretical concepts can be mapped into a concrete interpreter, this work serves as a resource for students to understand not only the theory of game semantics, but also λ-calculus, linear logic, and the π-calculus, specifically the session-typed variant. The implementation provides a hands-on experience that complements the theoretical study, making the concepts more accessible and easier to grasp.

6.2 Limitations

While YuugiLang successfully demonstrates the core ideas of game semantics, it is important to acknowledge its limitations. The language is intentionally minimal, and as such, it lacks many features that would be expected in a full-fledged programming language. For instance, it does not include features such as polymorphism, algebraic data types, or advanced control structures. The process layer is also quite basic, and does not support advanced concurrency primitives or sophisticated communication patterns, like multi-party sessions. The $ML_{||}$ implementation is also quite simple, and does not yet supports the concurrency on the expression layer, which is a key aspect of the π-calculus.

The set of features of Rust was not fully utilized. For instance, Rust's powerful macro system could be used to create more ergonomic syntax for defining processes and types. Leveraging the traits system could also allow for more flexible code, enabling polymorphism and code reuse. Rust's async/await syntax could be used to model asynchronous communication more naturally, improving the expressiveness of the process layer.

Additionaly, the implementation is a interpreter, rather than a compiler, which limits its performance and scalability. The focus was on clarity, rather than efficiency or optimization. If the goal was to create a production-ready language, many additional areas would need to be improved, such as error handling, debugging support, and tooling.

Finally, we did not explore the full range of theoretical results in game semantics, such as full abstraction or the relationship with other semantic models. The focus was on providing a practical implementation, rather than a comprehensive theoretical treatment.

6.3 Lessons learned

Developing YuugiLang has provided several valuable insights and lessons. One of the challenges was balancing the need for simplicity with the desire to accurately represent the theoretical concepts. It was important to carefully choose which features to include, and which to leave out, in order to maintain clarity without oversimplifying the ideas.

From the implementation side, working with Rust was a positive experience. Its strong type system and ownership model echo the principles of linear logic, making it a suitable choice for this project. However, the strictness of Rust's borrow checker sometimes made it difficult to implement rescursive or cyclic structures, requiring refactoring frequently to satisfy the compiler.

On the theoretical side, the project reinforced the importance of understanding the foundational concepts of game semantics, linear logic, and the π-calculus. It highlighted the connections between these areas, and how they can be integrated into a coherent framework for reasoning about computation and interaction.

Overall, the experience of developing YuugiLang has been both challenging and rewarding, providing a deeper understanding of both the theoretical and practical aspects of game semantics and programming language design.

6.4 Future work

Although we went for a minimal implementation, there are several directions that could be explored in future work to enhance and extend YuugiLang.

- *Language features:* both the payload and process layers could be enriched with additional features. For instance, adding polymorphism, algebraic data types, and more advanced control structures to the payload layer would increase its expressiveness. On the process layer, introducing more sophisticated concurrency primitives, such as multi-party sessions, could allow for more complex communication patterns.
- *Compilation:* developing a compiler for YuugiLang could improve its performance and scalability. This would involve translating the high-level constructs into lower-level representations, potentially targeting an intermediate language or even machine code. This would also open up opportunities for optimization and better resource management.
- *Verification:* integrating formal verification techniques could enhance the reliability of the language. This would involve model checking or theorem proving to ensure that programs adhere to specified properties, such as deadlock-freedom or type safety. It is also an avenue to explore game semantics for automated program analysis.

6.5 Closing toughts

YuugiLang is a small system, but it serves as a proof of concept that game semantics can be effectively integrated into a programming language. Its main contribution is educational, providing a concrete illustration of the abstract concepts of game semantics. While it has limitations, it opens up several interesting avenues for future exploration and development.

By continuing to build on this foundation, we can further explore the potential of game semantics as a framework for understanding and designing programming languages, ultimately contributing to the broader field of programming language theory and practice. We hope that this work can flourish into a more complete system, and that it inspires others to explore the rich interplay between games, logic, and computation.

References

Barendregt, H. P. (1984). *The lambda calculus: Its syntax and semantics*. North-Holland.

Caires, L., Pfenning, F., & Toninho, B. (2016). Linear logic propositions as session types. *Mathematical Structures in Computer Science*, *26*(6), 367–423.

Castellan, S., Clairambault, P., & Winskel, G. (2017). Games and strategies as event structures. *Logical Methods in Computer Science*, *13*(3), 1–49.

Castellan, S., Stefanesco, L., & Yoshida, N. (2020). Game semantics: Easy as pi. *arXiv*. Available on: https://arxiv.org/abs/2011.05248,

Fernández, M., & Mackie, I. An introduction to linear logic. In: In *Theoretical computer science. 284*. (1). Elsevier, 2002, 1–15.

Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, *39*, 176–210.

Ghica, D. R. (2009). Applications of game semantics: From program analysis to hardware syntesis. *24th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 17–26.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, *50*(1), 1–102.

Hindley, J. R. (1997). *Basic type theory*. Cambridge University Press.

Hyland, J. M. E., & Ong, C.-H. L. (2000). On full abstraction for PCF: I, II, and III. *Information and Computation*, *163*(2), 285–408.

Ivanov, N. (2022). Is rust C++fast? benchmrking system languages on everyday routines. Available on: https://arxiv.org/pdf/2209.09127,

Klabnik, S., & Nichols, C. (2019). *The rust programming language*. No Starch Press. Available on: https://doc.rust-lang.org/book/,

Laird, J. (2005). A game semantics of the asyncronous π-calculus. In M. Abadi & L. Alfaro (Eds.), *Concur 2005 — concurrency theory: 16th international conference, concur 2005* (pp. 51–65). Springer. https://doi.org/10.1007/11539452

Milner, R. (1999). *Communicating and mobile systems: The pi-calculus*. Cambridge University Press.

Milner, R., Parrow, J., & Walker, D. (1992). A calculus of communicating systems. *Information and Computation*, *100*(1), 1–77.

Sakayori, K., & Yoshida, N. (2017). A truly concurrent game model of the π-calculus. *Foundations of Software Science and Computation Structures*, 389–406. https://doi.org/10.1007/978-3-662-54458-7

Sørensen, M., & Urzyczyn, P. (2006). *Lectures on the curry-howard isomorphism*. Springer.

Vasconcelos, V. T. (2009). Fundamentals of session types. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Formal Methods for Mobile, Concurrent, and Distributed Systems*, 158–286.

Wadler, P. (1993). A taste of linear logic. *Mathematical Foundations of Computer Science 1993*, 185–210.

Wadler, P. (2012). Propositions as sessions. *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 273–286.

Page left blank intentionally.

Appendix A
Implementation

This appendix provides the source code of the implementation of YuugiLang, as described in chapters Chapter 4 and Chapter 5. The code is divided into the following files:

- `syntax.rs`: the abstract syntax tree of YuugiLang;
- `channel.rs`: the structure of channels;
- `parser.rs`: implementation of the parser, from source code to AST;
- `game_semantics.rs`: the evaluation process, implementing the dynamic semantics;
- `runtime.rs`: the type checking and runtime execution structs and functions;
- `error.rs`: error handling;
- `main.rs`: the main file, which ties everything together and provides the command line interface.

1.1 Syntax

```rust
use crate::{
    channel::{ChannelInternal, SessionAction},
    error::MemoryError::{self, *},
};
use std::{
    collections::HashMap,
    fmt::{Display, Formatter},
    sync::{Arc, Mutex},
};

#[derive(Debug, Clone, PartialEq)]
pub enum Type {
    Unit,
    Int,
    Bool,
    RefType(Box<Type>),
    Function(Box<Type>, Box<Type>),
    Channel(Box<Type>),
    Session(Box<SessionAction>),
    Rec(String, Box<Type>),
}

impl std::fmt::Display for Type {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        use Type::*;
        match self {
            Unit => write!(f, "0"),
            Int => write!(f, "Int"),
            Bool => write!(f, "Bool"),
            RefType(t) => write!(f, "Ref<{t}>"),
            Function(t1, t2) => write!(f, "({t1} -> {t2})"),
```

```
32              Channel(t) => write!(f, "Channel<{t}>"),
33              Session(a) => write!(f, "Session({a:?})"),
34              Rec(var, t) => write!(f, "rec {var}. {t}"),
35          }
36      }
37  }
38
39  #[derive(Debug, Clone)]
40  pub enum Value {
41      Unit,
42      Var(String),
43      Lambda {
44          name: Option<String>,
45          param: String,
46          param_type: Option<Type>,
47          body: Box<CmlTerm>,
48      },
49      Int(i64),
50      Address(usize),
51      Bool(bool),
52      Label(String),
53      Channel(Arc<Mutex<ChannelInternal>>),
54  }
55
56  impl PartialEq for Value {
57      fn eq(&self, other: &Self) -> bool {
58          use Value::*;
59          match (self, other) {
60              (Unit, Unit) => true,
61              (Var(n1), Var(n2)) => n1 == n2,
62              (
63                  Lambda {
64                      name: s1,
65                      param: p1,
66                      param_type: t1,
67                      body: b1,
68                  },
69                  Lambda {
70                      name: s2,
71                      param: p2,
72                      param_type: t2,
73                      body: b2,
74                  },
75              ) => {
76                  if let (Some(t1), Some(t2)) = (t1, t2) {
77                      s1 == s2 && p1 == p2 && t1 == t2 && b1 == b2
78                  } else {
79                      s1 == s2 && p1 == p2 && b1 == b2
80                  }
81              }
82              (Int(i1), Int(i2)) => i1 == i2,
83              (Address(a1), Address(a2)) => a1 == a2,
84              (Bool(b1), Bool(b2)) => b1 == b2,
85              (Label(l1), Label(l2)) => l1 == l2,
86              (Channel(c1), Channel(c2)) => Arc::ptr_eq(c1, c2),
87              _ => false,
88          }
89      }
90  }
91
92  impl Display for Value {
93      fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
94          use Value::*;
95          match self {
96              Var(name) => write!(f, "{name}"),
97              Lambda {
98                  name: self_name,
99                  param,
```

```
                param_type,
                body,
            } ⇒ {
                if let Some(name) = self_name {
                    if let Some(param_type) = param_type {
                        write!(f, "λ {name}: ({param} -> {param_type}). ({body})")
                    } else {
                        write!(f, "λ {name}: ({param}). ({body})")
                    }
                } else if let Some(param_type) = param_type {
                    write!(f, "λ ({param}: {param_type}). ({body})")
                } else {
                    write!(f, "λ ({param}). ({body})")
                }
            }
            Address(addr) ⇒ write!(f, "Addr({addr})"),
            Int(i) ⇒ write!(f, "{i}"),
            Bool(b) ⇒ write!(f, "{b}"),
            Unit ⇒ write!(f, "()"),
            Channel(internal) ⇒ write!(f, "Channel({})", internal.lock().unwrap().name),
            Label(label) ⇒ {
                write!(f, "Label({label})",)
            }
        }
    }
}

#[derive(Debug, Clone, PartialEq)]
pub enum CmlTerm {
    Bottom,
    Val(Value),
    App(Box<CmlTerm>, Box<CmlTerm>),
    Fix(String, Option<(Type, Type)>, Box<CmlTerm>),
    If(Box<CmlTerm>, Box<CmlTerm>, Box<CmlTerm>),
    Add(Box<CmlTerm>, Box<CmlTerm>),
    Equal(Box<CmlTerm>, Box<CmlTerm>),
    Get(Box<CmlTerm>),
    Set(Box<CmlTerm>, Box<CmlTerm>),
    Ref(Box<CmlTerm>),
    Annot(Box<CmlTerm>, Type),
}

#[derive(Debug, Clone, PartialEq)]
pub enum SessionTerm {
    NewChannel(String, Option<SessionAction>, Box<SessionTerm>),
    Send(CmlTerm, CmlTerm, Box<SessionTerm>),
    Receive(CmlTerm, String, Box<SessionTerm>),
    Select(CmlTerm, CmlTerm, Box<SessionTerm>),
    Offer(CmlTerm, Vec<(CmlTerm, Box<SessionTerm>)>),
    Parallel(Box<SessionTerm>, Box<SessionTerm>),
    SVar(String),
    Replicate(Box<SessionTerm>),
    Rec(String, Box<SessionTerm>),
    Link(CmlTerm, CmlTerm, Box<SessionTerm>),
    End,
}

impl Display for CmlTerm {
    fn fmt(&self, fmt: &mut Formatter<'_>) -> std::fmt::Result {
        use CmlTerm::*;
        match self {
            Val(v) ⇒ write!(fmt, "{v}"),
            App(t1, t2) ⇒ write!(fmt, "apply ({t1}) ({t2})"),
            Fix(var, opt_type, body) ⇒ {
                if let Some((param_type, ret_type)) = opt_type {
                    write!(fmt, "fix {var}: ({param_type} -> {ret_type}). ({body})")
                } else {
                    write!(fmt, "fix {var}. ({body})")
```

```
                }
            }
            Bottom ⇒ write!(fmt, "□"),
            If(cond, t, f) ⇒ write!(fmt, "if ({cond}) then ({t}) else ({f})"),
            Add(lhs, rhs) ⇒ write!(fmt, "({lhs}) + ({rhs})"),
            Equal(lhs, rhs) ⇒ write!(fmt, "({lhs}) == ({rhs})"),
            Get(t) ⇒ write!(fmt, "get ({t})"),
            Set(t1, t2) ⇒ write!(fmt, "set ({t1}) ({t2})"),
            Ref(t) ⇒ write!(fmt, "ref ({t})"),
            Annot(t, ty) ⇒ write!(fmt, "({t} : {ty})"),
        }
    }
}

impl Display for SessionTerm {
    fn fmt(&self, fmt: &mut Formatter<'_>) -> std::fmt::Result {
        use SessionTerm::*;
        match self {
            NewChannel(name, opt_sa, body) ⇒ {
                if let Some(sa) = opt_sa {
                    write!(fmt, "new {name}: {sa} in {body}")
                } else {
                    write!(fmt, "new {name} in {body}")
                }
            }
            Send(t1, t2, cont) ⇒ write!(fmt, "({t1}) !> ({t2}).{cont}"),
            Receive(t1, name, cont) ⇒ write!(fmt, "({t1}) <({name})>.{cont}"),
            Parallel(t1, t2) ⇒ write!(fmt, "({t1}) || ({t2})"),
            Replicate(t) ⇒ write!(fmt, "repl({t})"),
            Select(channel, label, cont) ⇒ {
                write!(fmt, "select {label} on {channel}.{cont}")
            }
            Offer(channel, branches) ⇒ {
                let branches_str = branches
                    .iter()
                    .map(|(val, term)| format!("{val} ⇒ {term}"))
                    .collect::<Vec<_>>()
                    .join(", ");
                write!(fmt, "offer {channel} {{ {branches_str} }}")
            }
            SVar(name) ⇒ write!(fmt, "{name}"),
            Rec(name, body) ⇒ write!(fmt, "rec {} ({})", name.clone(), body),
            Link(t1, t2, cont) ⇒ write!(fmt, "{t1} <-> {t2}.{cont}"),
            End ⇒ write!(fmt, "0"),
        }
    }
}

pub type Env = Vec<HashMap<String, Value>>;

pub fn lookup_var(env: &Env, name: &str) -> Option<Value> {
    for scope in env.iter().rev() {
        if let Some(val) = scope.get(name) {
            return Some(val.clone());
        }
    }
    None
}

pub fn insert_at_new_scope(env: &mut Env, name: String, value: Value) {
    env.push(HashMap::new());
    env.last_mut().unwrap().insert(name, value);
}

pub fn insert_var(env: &mut Env, name: String, value: Value) {
    if let Some(scope) = env.last_mut() {
        scope.insert(name, value);
    } else {
```

```
        insert_at_new_scope(env, name, value);
    }
}

pub fn insert_split(env: &mut Env, v1: Value, v2: Value) {
    env.push(HashMap::new());
    env.last_mut().unwrap().insert("_left".into(), v1);
    env.last_mut().unwrap().insert("_right".into(), v2);
}

#[derive(Debug, Clone, Default)]
pub struct Memory {
    pub heap: Vec<Value>,
}

impl Memory {
    pub fn new() -> Self {
        Memory { heap: Vec::new() }
    }

    pub fn alloc(&mut self, value: Value) -> usize {
        self.heap.push(value);
        self.heap.len() - 1
    }

    pub fn get(&self, addr: usize) -> Result<&Value, MemoryError> {
        if addr ≥ self.heap.len() {
            return Err(AddressNotFound(addr));
        }
        Ok(&self.heap[addr])
    }

    pub fn set(&mut self, addr: usize, value: Value) -> Result<(), MemoryError> {
        if addr ≥ self.heap.len() {
            return Err(AddressNotFound(addr));
        }
        self.heap[addr] = value;
        Ok(())
    }
}
```

1.2 Channel

```
use crate::{
    error::{
        MemoryError::*,
        SessionError::{self, *},
        TypeError::*,
    },
    game_semantics::Process,
    syntax::{CmlTerm, SessionTerm, Type, Value},
};
use std::{
    collections::{HashMap, HashSet, VecDeque},
    error::Error,
    fmt::{Display, Formatter},
    sync::{Arc, Mutex, Weak},
};

#[derive(Debug, Clone, Default)]
pub struct ChannelInternal {
    pub name: String,
    pub queue: VecDeque<Value>,
    pub machine: ProtocolMachine,
    pub linked_to: Option<Weak<Mutex<ChannelInternal>>>,
    pub composed: bool,
```

```
24      pub restricted: bool,
25      pub creation_node: Option<usize>,
26      pub last_send_node: Option<usize>,
27      pub last_receive_node: Option<usize>,
28  }
29
30  impl PartialEq for ChannelInternal {
31      fn eq(&self, other: &Self) -> bool {
32          self.name == other.name
33              && self.queue == other.queue
34              && self.machine == other.machine
35              && self.composed == other.composed
36      }
37  }
38
39  impl ChannelInternal {
40      pub fn new(name: &str) -> Self {
41          ChannelInternal {
42              name: name.to_string(),
43              queue: VecDeque::new(),
44              machine: ProtocolMachine::default(),
45              linked_to: None,
46              composed: false,
47              restricted: false,
48              creation_node: None,
49              last_send_node: None,
50              last_receive_node: None,
51          }
52      }
53
54      pub fn with_protocol(name: &str, action: SessionAction) -> Self {
55          ChannelInternal {
56              name: name.to_string(),
57              queue: VecDeque::new(),
58              machine: ProtocolMachine::new(action),
59              linked_to: None,
60              composed: false,
61              restricted: false,
62              creation_node: None,
63              last_send_node: None,
64              last_receive_node: None,
65          }
66      }
67
68      pub fn restricted(name: &str, creation_node: usize) -> Self {
69          ChannelInternal {
70              name: name.to_string(),
71              queue: VecDeque::new(),
72              machine: ProtocolMachine::default(),
73              linked_to: None,
74              composed: false,
75              restricted: true,
76              creation_node: Some(creation_node),
77              last_send_node: None,
78              last_receive_node: None,
79          }
80      }
81
82      pub fn restricted_with_action(name: &str, action: SessionAction, creation_node: usize) -> Self {
83          ChannelInternal {
84              name: name.to_string(),
85              queue: VecDeque::new(),
86              machine: ProtocolMachine::new(action),
87              linked_to: None,
88              composed: false,
89              restricted: true,
90              creation_node: Some(creation_node),
91              last_send_node: None,
```

```
92              last_receive_node: None,
93          }
94      }
95
96      pub fn send(chan: &Arc<Mutex<ChannelInternal>>, value: Value) -> Result<(), Box<dyn Error + Send + Sync
97          let mut c = chan.lock().unwrap();
98          let expected_type = c.machine.current.expected_type()?;
99
100         match &c.machine.current {
101             SessionAction::SendAction(t, _) if *t == expected_type => {}
102             SessionAction::SendAction(t, _) => {
103                 return Err(Box::new(TypeMismatch(expected_type, t.clone())));
104             }
105             other => {
106                 return Err(Box::new(InvalidTransition(
107                     other.clone(),
108                     SessionAction::End,
109                 )));
110             }
111         }
112
113         let peer_action = c.machine.current.dual();
114
115         if c.composed {
116             let peer_arc = c
117                 .linked_to
118                 .as_ref()
119                 .and_then(|w| w.upgrade())
120                 .ok_or_else(|| Box::new(DisconnectedChannel(c.name.clone())))?;
121
122             drop(c);
123
124             let (first, second) = if Arc::as_ptr(chan) <= Arc::as_ptr(&peer_arc) {
125                 (chan.clone(), peer_arc.clone())
126             } else {
127                 (peer_arc.clone(), chan.clone())
128             };
129
130             let mut c1 = first.lock().unwrap();
131             let mut c2 = second.lock().unwrap();
132
133             let (c_guard, peer_guard) = if Arc::as_ptr(chan) <= Arc::as_ptr(&peer_arc) {
134                 (&mut c1, &mut c2)
135             } else {
136                 (&mut c2, &mut c1)
137             };
138
139             c_guard
140                 .machine
141                 .linked_transition(peer_action.clone(), &mut peer_guard.machine)?;
142
143             peer_guard.queue.push_back(value);
144             return Ok(());
145         }
146
147         c.machine.transition(peer_action)?;
148         c.queue.push_back(value);
149         Ok(())
150     }
151
152     pub fn receive(chan: &Arc<Mutex<ChannelInternal>>) -> Result<Value, Box<dyn Error + Send + Sync>> {
153         let mut c = chan.lock().unwrap();
154
155         let _expected_type = match &c.machine.current {
156             SessionAction::ReceiveAction(t, _) => t.clone(),
157             other => {
158                 return Err(Box::new(InvalidTransition(
159                     other.clone(),
```

```
                    SessionAction::End,
                )));
            }
        };

        let peer_action = c.machine.current.dual();

        if c.composed {
            let peer_arc = c
                .linked_to
                .as_ref()
                .and_then(|w| w.upgrade())
                .ok_or_else(|| Box::new(DisconnectedChannel(c.name.clone())))?;

            drop(c);

            let (first, second) = if Arc::as_ptr(chan) ≤ Arc::as_ptr(&peer_arc) {
                (chan.clone(), peer_arc.clone())
            } else {
                (peer_arc.clone(), chan.clone())
            };

            let mut c1 = first.lock().unwrap();
            let mut c2 = second.lock().unwrap();

            let (c_guard, peer_guard) = if Arc::as_ptr(chan) ≤ Arc::as_ptr(&peer_arc) {
                (&mut c1, &mut c2)
            } else {
                (&mut c2, &mut c1)
            };

            c_guard
                .machine
                .linked_transition(peer_action.clone(), &mut peer_guard.machine)?;

            let value = c_guard
                .queue
                .pop_front()
                .ok_or_else(|| Box::new(EmptyChannel(c_guard.name.clone())))?;

            return Ok(value);
        }

        c.machine.transition(peer_action)?;
        let value = c
            .queue
            .pop_front()
            .ok_or_else(|| Box::new(EmptyChannel(c.name.clone())))?;
        Ok(value)
    }

    pub fn handle_select(
        channel: &Arc<Mutex<ChannelInternal>>,
        label: Value,
        continuation: SessionTerm,
        process: &mut Process,
    ) -> Result<(), Box<dyn Error + Send + Sync>> {
        let lbl = match label {
            Value::Label(s) ⇒ s,
            _ ⇒ return Err(Box::new(ExpectedSessionChoice)),
        };

        let peer_opt = {
            let mut c = channel.lock().unwrap();
            let arms = match &c.machine.current {
                SessionAction::SelectAction(arms) ⇒ arms,
                _ ⇒ return Err(Box::new(ExpectedSessionChannel)),
            };
```

```
228
229             if !arms.iter().any(|(l, _)| l == &lbl) {
230                 return Err(Box::new(InvalidBranchSelection(
231                     SessionAction::SelectAction(arms.clone()),
232                     SessionAction::Var(lbl.clone()),
233                 )));
234             }
235
236             c.machine.choose_label(&lbl)?;
237             c.linked_to.as_ref().and_then(|w| w.upgrade())
238         };
239
240         if let Some(peer) = peer_opt {
241             peer.lock().unwrap().queue.push_back(Value::Label(lbl.clone()));
242         } else {
243             channel.lock().unwrap().queue.push_back(Value::Label(lbl.clone()));
244         }
245
246         process.term = continuation;
247         Ok(())
248     }
249
250     pub fn handle_offer(
251         channel: &Arc<Mutex<ChannelInternal>>,
252         branches: Vec<(Value, SessionTerm)>,
253         process: &mut Process,
254     ) -> Result<Value, Box<dyn Error + Send + Sync>> {
255         let (lbl, _peer_opt) = {
256             let mut c = channel.lock().unwrap();
257
258             let label = c
259                 .queue
260                 .pop_front()
261                 .ok_or_else(|| EmptyChannel(c.name.clone()))?;
262             let lbl = match &label {
263                 Value::Label(s) => s.clone(),
264                 _ => return Err(Box::new(ExpectedSessionChoice)),
265             };
266
267             match &c.machine.current {
268                 SessionAction::BranchAction(arms) => {
269                     if !arms.iter().any(|(l, _)| l == &lbl) {
270                         return Err(Box::new(InvalidBranchSelection(
271                             SessionAction::BranchAction(arms.clone()),
272                             SessionAction::Var(lbl.clone()),
273                         )));
274                     }
275                 }
276                 _ => return Err(Box::new(ExpectedSessionChannel)),
277             }
278             c.machine.accept_label(&lbl)?;
279             (lbl, c.linked_to.as_ref().and_then(|w| w.upgrade()))
280         };
281
282         let idx = branches.iter().position(|(v, _)| matches!(v, Value::Label(l) if l == &lbl))
283             .ok_or_else(|| Box::new(ValueNotInBranch(CmlTerm::Val(Value::Label(lbl.clone())))))?;
284         let (_, cont) = branches.into_iter().nth(idx).unwrap();
285
286         process.term = cont;
287         Ok(Value::Label(lbl))
288     }
289
290     pub fn mark_composed(&mut self, other: &Weak<Mutex<ChannelInternal>>) {
291         self.composed = true;
292         self.linked_to = Some(other.clone());
293     }
294
295     pub fn link_channels(
```

```
296         chan1: &Arc<Mutex<ChannelInternal>>,
297         chan2: &Arc<Mutex<ChannelInternal>>,
298     ) -> Result<(), Box<dyn Error + Send + Sync>> {
299         let (a, b) = if Arc::as_ptr(chan1) ≤ Arc::as_ptr(chan2) {
300             (chan1, chan2)
301         } else {
302             (chan2, chan1)
303         };
304 
305         let mut c1 = a.lock().unwrap();
306         let mut c2 = b.lock().unwrap();
307 
308         ProtocolMachine::validate_composition(&c1.machine, &c2.machine)?;
309 
310         let w1 = Arc::downgrade(chan1);
311         let w2 = Arc::downgrade(chan2);
312 
313         c1.mark_composed(&w2);
314         c2.mark_composed(&w1);
315 
316         Ok(())
317     }
318 
319     pub fn validate_final_state(&self) -> Result<(), Box<dyn Error>> {
320         if !self.queue.is_empty() {
321             return Err(Box::new(ChannelPending(self.name.clone())));
322         }
323 
324         if let SessionAction::End = self.machine.current {
325             Ok(())
326         } else {
327             Err(Box::new(IncompleteSession(
328                 self.name.clone(),
329                 self.machine.current.clone(),
330             )))
331         }
332     }
333 }
334 
335 #[derive(Debug, Clone, Default, PartialEq)]
336 pub struct ProtocolMachine {
337     pub current: SessionAction,
338     pub stack: Vec<SessionAction>,
339     pub history: Vec<SessionAction>,
340 }
341 
342 impl ProtocolMachine {
343     pub fn new(initial: SessionAction) -> Self {
344         ProtocolMachine {
345             current: initial,
346             stack: Vec::new(),
347             history: Vec::new(),
348         }
349     }
350 
351     pub fn choose_label(&mut self, lbl: &str) -> Result<(), Box<dyn Error + Send + Sync>> {
352         match &self.current {
353             SessionAction::SelectAction(arms) ⇒ {
354                 if let Some((_, next)) = arms.iter().find(|(l, _)| l == lbl) {
355                     self.history.push(self.current.clone());
356                     self.current = next.clone();
357                     Ok(())
358                 } else {
359                     Err(Box::new(InvalidBranchLabel(lbl.to_string())))
360                 }
361             }
362             _ ⇒ Err(Box::new(ExpectedSelect)),
363         }
```

```
364     }
365
366     pub fn accept_label(&mut self, lbl: &str) -> Result<(), Box<dyn Error + Send + Sync>> {
367         match &self.current {
368             SessionAction::BranchAction(arms) => {
369                 if let Some((_, next)) = arms.iter().find(|(l, _)| l == lbl) {
370                     self.history.push(self.current.clone());
371                     self.current = next.clone();
372                     Ok(())
373                 } else {
374                     Err(Box::new(InvalidBranchLabel(lbl.to_string())))
375                 }
376             }
377             _ => Err(Box::new(ExpectedOffer)),
378         }
379     }
380
381     pub fn validate_transition(&self, action: SessionAction) -> Result<(), Box<dyn Error + Send + Sync>> {
382         if !self.current.is_dual_of(&action) {
383             return Err(Box::new(InvalidTransition(self.current.clone(), action)));
384         }
385         Ok(())
386     }
387
388     fn process_state(&mut self, peer_action: SessionAction) -> SessionAction {
389         use SessionAction::*;
390         let new_state = match (&self.current, &peer_action) {
391             (SendAction(_, cont), ReceiveAction(_, _)) => (**cont).clone(),
392             (ReceiveAction(_, cont), SendAction(_, _)) => (**cont).clone(),
393             (RecAction(_, body), _) => (**body).clone(),
394             _ => self.current.clone(),
395         };
396         self.history.push(self.current.clone());
397         self.current = new_state.clone();
398         new_state
399     }
400
401     pub fn transition(&mut self, peer_action: SessionAction) -> Result<(), Box<dyn Error + Send + Sync>> {
402         self.validate_transition(peer_action.clone())?;
403         let _ = self.process_state(peer_action);
404         Ok(())
405     }
406
407     pub fn linked_transition(
408         &mut self,
409         peer_action: SessionAction,
410         linked_machine: &mut ProtocolMachine,
411     ) -> Result<(), Box<dyn Error + Send + Sync>> {
412         self.validate_transition(peer_action.clone())?;
413         linked_machine.validate_transition(peer_action.dual())?;
414
415         let _ = self.process_state(peer_action.clone());
416         let _ = linked_machine.process_state(peer_action.dual());
417
418         Ok(())
419     }
420
421     pub fn validate_continuation(&self, cont: &SessionTerm) -> Result<(), Box<dyn Error + Send + Sync>> {
422         use SessionAction::*;
423         match (&self.current, cont) {
424             (SendAction(_, _), SessionTerm::Send(_, _, _)) => Ok(()),
425             (ReceiveAction(_, _), SessionTerm::Receive(_, _, _)) => Ok(()),
426             (SelectAction(_), SessionTerm::Select(_, _, _)) => Ok(()),
427             (BranchAction(_), SessionTerm::Offer(_, _)) => Ok(()),
428             _ => Err(Box::new(InvalidContinuation)),
429         }
430     }
431
```

```
432     pub fn validate_composition(
433         machine1: &ProtocolMachine,
434         machine2: &ProtocolMachine,
435     ) -> Result<(), SessionError> {
436         machine1
437             .current
438             .validate_duality_all_paths(&machine2.current)
439     }
440
441     pub fn step_with_peer_action(
442         &mut self,
443         peer_action: SessionAction,
444     ) -> Result<SessionAction, Box<dyn Error + Send + Sync>> {
445         self.validate_transition(peer_action.clone())?;
446         Ok(self.process_state(peer_action))
447     }
448 }
449
450 #[derive(Debug, Clone, PartialEq, Default)]
451 pub enum SessionAction {
452     Var(String),
453     SendAction(Type, Box<SessionAction>),
454     ReceiveAction(Type, Box<SessionAction>),
455     SelectAction(Vec<(String, SessionAction)>),
456     BranchAction(Vec<(String, SessionAction)>),
457     RecAction(String, Box<SessionAction>),
458     #[default]
459     End,
460 }
461
462 impl Display for SessionAction {
463     fn fmt(&self, fmt: &mut Formatter<'_>) -> std::fmt::Result {
464         use SessionAction::*;
465         match self {
466             Var(name) => write!(fmt, "{name}"),
467             SendAction(t, c) => write!(fmt, "!>{t}. {c}"),
468             ReceiveAction(t, c) => write!(fmt, "?<{t}. {c}"),
469             SelectAction(actions) => {
470                 let actions_str: Vec<String> = actions
471                     .iter()
472                     .map(|(l, s)| format!("{}: {}", l, s))
473                     .collect();
474                 write!(fmt, "⊕{{{}}}", actions_str.join(", "))
475             }
476             BranchAction(actions) => {
477                 let actions_str: Vec<String> = actions
478                     .iter()
479                     .map(|(l, s)| format!("{}: {}", l, s))
480                     .collect();
481                 write!(fmt, "&{{{}}}", actions_str.join(", "))
482             }
483             RecAction(name, body) => write!(fmt, "μ {name}. {body}"),
484             End => write!(fmt, "end"),
485         }
486     }
487 }
488
489 impl SessionAction {
490     pub fn dual(&self) -> Self {
491         use SessionAction::*;
492         match self {
493             Var(name) => Var(name.clone()),
494             SendAction(t, c) => ReceiveAction(t.clone(), Box::new(c.dual())),
495             ReceiveAction(t, c) => SendAction(t.clone(), Box::new(c.dual())),
496             SelectAction(actions) => {
497                 BranchAction(actions.iter().map(|(l, s)| (l.clone(), s.dual())).collect())
498             }
499             BranchAction(actions) => {
```

```
                SelectAction(actions.iter().map(|(l, s)| (l.clone(), s.dual())).collect())
            }
            RecAction(name, body) => RecAction(name.clone(), Box::new(body.dual())),
            End => End,
        }
    }

    pub fn is_dual_of(&self, other: &SessionAction) -> bool {
        use SessionAction::*;
        match (self, other) {
            (End, End) => true,

            (Var(a), Var(b)) => a == b,

            (SendAction(t1, c1), ReceiveAction(t2, c2))
            | (ReceiveAction(t1, c1), SendAction(t2, c2)) => t1 == t2 && c1.is_dual_of(c2),

            (RecAction(x1, b1), RecAction(x2, b2)) => x1 == x2 && b1.is_dual_of(b2),

            (SelectAction(a1), BranchAction(a2)) | (BranchAction(a1), SelectAction(a2)) => {
                if a1.len() != a2.len() {
                    return false;
                }
                let m1: HashMap<_, _> = a1.iter().map(|(l, s)| (l, s)).collect();
                let m2: HashMap<_, _> = a2.iter().map(|(l, s)| (l, s)).collect();
                m1.len() == m2.len()
                    && m1
                        .iter()
                        .all(|(l, s1)| m2.get(l).is_some_and(|s2| s1.is_dual_of(s2)))
            }

            _ => false,
        }
    }

    pub fn next_state(self) -> Self {
        use SessionAction::*;
        match self {
            Var(x) => Var(x),
            RecAction(_, body) => *body,
            SendAction(_, cont) | ReceiveAction(_, cont) => cont.dual(),
            SelectAction(mut xs) | BranchAction(mut xs) => {
                if let Some((_, s)) = xs.drain(..1).next() {
                    s.dual()
                } else {
                    End
                }
            }
            End => End,
        }
    }

    pub fn unfold_recursive(&mut self, name: &str, replacement: &SessionAction) {
        use SessionAction::*;
        match self {
            Var(n) if n == name => *self = replacement.clone(),
            RecAction(n, body) if n == name => body.unfold_recursive(name, replacement),

            SendAction(_, cont) | ReceiveAction(_, cont) => {
                cont.unfold_recursive(name, replacement)
            }

            SelectAction(v) | BranchAction(v) => {
                for (_, s) in v.iter_mut() {
                    s.unfold_recursive(name, replacement);
                }
            }

```

```
568             _ => {}
569         }
570     }
571
572     pub fn can_unfold(&self, name: &str) -> bool {
573         use SessionAction::*;
574         match self {
575             RecAction(n, _) if n == name => true,
576             SendAction(_, c) | ReceiveAction(_, c) => c.can_unfold(name),
577             SelectAction(v) | BranchAction(v) => v.iter().any(|(_, s)| s.can_unfold(name)),
578             _ => false,
579         }
580     }
581
582     pub fn allows_select(&self) -> bool {
583         use SessionAction::*;
584         match self {
585             SelectAction(_) => true,
586             SendAction(_, c) | ReceiveAction(_, c) | RecAction(_, c) => c.allows_select(),
587             BranchAction(v) => v.iter().any(|(_, s)| s.allows_select()),
588             _ => false,
589         }
590     }
591
592     pub fn allows_offer(&self) -> bool {
593         use SessionAction::*;
594         match self {
595             BranchAction(_) => true,
596             SendAction(_, c) | ReceiveAction(_, c) | RecAction(_, c) => c.allows_offer(),
597             SelectAction(v) => v.iter().any(|(_, s)| s.allows_offer()),
598             _ => false,
599         }
600     }
601
602     pub fn expected_type(&self) -> Result<Type, Box<dyn Error + Send + Sync>> {
603         use SessionAction::*;
604         match self {
605             SendAction(t, _) | ReceiveAction(t, _) => Ok(t.clone()),
606             _ => Err(Box::new(ExpectedSendOrReceive)),
607         }
608     }
609
610     fn dfs_validate_duality(
611         &self,
612         other: &SessionAction,
613         seen: &mut HashSet<(usize, usize)>,
614     ) -> Result<(), SessionError> {
615         use SessionAction::*;
616
617         let key = (self as *const _ as usize, other as *const _ as usize);
618         if !seen.insert(key) {
619             return Ok(());
620         }
621
622         match (self, other) {
623             (End, End) => Ok(()),
624
625             (Var(_), Var(_)) => Ok(()),
626
627             (SendAction(t1, c1), ReceiveAction(t2, c2))
628             | (ReceiveAction(t1, c1), SendAction(t2, c2)) => {
629                 if t1 != t2 {
630                     return Err(InvalidComposition(self.clone(), other.clone()));
631                 }
632                 c1.dfs_validate_duality(c2, seen)
633             }
634
635             (RecAction(x1, b1), RecAction(x2, b2)) if x1 == x2 => b1.dfs_validate_duality(b2, seen),
```

```
636
637             (SelectAction(xs), BranchAction(ys)) | (BranchAction(xs), SelectAction(ys)) ⇒ {
638                 let mx: HashMap<_, _> = xs.iter().cloned().collect();
639                 let my: HashMap<_, _> = ys.iter().cloned().collect();
640
641                 if mx.len() ≠ my.len() || !mx.keys().all(|k| my.contains_key(k)) {
642                     return Err(BranchCountMismatch(mx.len(), my.len()));
643                 }
644
645                 for (lbl, sx) in mx {
646                     sx.dfs_validate_duality(my.get(&lbl).unwrap(), seen)?
647                 }
648
649                 Ok(())
650             }
651
652             _ ⇒ Err(InvalidComposition(self.clone(), other.clone())),
653         }
654     }
655
656     pub fn validate_duality_all_paths(&self, other: &SessionAction) -> Result<(), SessionError> {
657         let mut seen = HashSet::new();
658         self.dfs_validate_duality(other, &mut seen)
659     }
660 }
661
662 #[cfg(test)]
663 pub mod test {
664     use crate::channel::ProtocolMachine;
665     #[test]
666     fn send_receive_advances_both_sides() {
667         use crate::channel::SessionAction::*;
668         use crate::syntax::Type;
669
670         let s = SendAction(Type::Int, Box::new(End));
671         let r = ReceiveAction(Type::Int, Box::new(End));
672
673         let mut m1 = ProtocolMachine::new(s.clone()); // local expects to send
674         let mut m2 = ProtocolMachine::new(r.clone()); // peer expects to receive
675
676         // Apply peer events: m1 sees peer Receive, m2 sees peer Send
677         m1.transition(ReceiveAction(Type::Int, Box::new(End)))
678             .unwrap();
679         m2.transition(SendAction(Type::Int, Box::new(End))).unwrap();
680
681         assert_eq!(m1.current, End);
682         assert_eq!(m2.current, End);
683     }
684
685     #[test]
686     fn select_offer_picks_label_continuation() {
687         use crate::channel::SessionAction::*;
688         let s = SelectAction(vec![
689             ("L".into(), End),
690             ("R".into(), RecAction("X".into(), Box::new(End))),
691         ]);
692         let b = s.dual();
693
694         let mut m_sel = ProtocolMachine::new(s);
695         let mut m_off = ProtocolMachine::new(b);
696
697         m_sel.choose_label("R").unwrap();
698         m_off.accept_label("R").unwrap();
699
700         assert_eq!(m_sel.current, RecAction("X".into(), Box::new(End)));
701         assert_eq!(m_off.current, RecAction("X".into(), Box::new(End)));
702     }
703
```

```
    #[test]
    fn channel_select_offer_flow() {
        use crate::channel::{ChannelInternal, SessionAction::*};
        use std::sync::{Arc, Mutex};

        let c1 = Arc::new(Mutex::new(ChannelInternal::with_protocol(
            "c1",
            SelectAction(vec![("A".into(), End), ("B".into(), End)]),
        )));
        let c2 = Arc::new(Mutex::new(ChannelInternal::with_protocol(
            "c2",
            BranchAction(vec![("A".into(), End), ("B".into(), End)]),
        )));

        ChannelInternal::link_channels(&c1, &c2).unwrap();

        let mut p1 = crate::game_semantics::Process::new(crate::syntax::SessionTerm::End);
        let mut p2 = crate::game_semantics::Process::new(crate::syntax::SessionTerm::End);

        ChannelInternal::handle_select(
            &c1,
            crate::syntax::Value::Label("B".into()),
            crate::syntax::SessionTerm::End,
            &mut p1,
        )
        .unwrap();
        let got = ChannelInternal::handle_offer(
            &c2,
            vec![
                (
                    crate::syntax::Value::Label("A".into()),
                    crate::syntax::SessionTerm::End,
                ),
                (
                    crate::syntax::Value::Label("B".into()),
                    crate::syntax::SessionTerm::End,
                ),
            ],
            &mut p2,
        )
        .unwrap();

        assert_eq!(got, crate::syntax::Value::Label("B".into()));
        assert!(matches!(c1.lock().unwrap().machine.current, End));
        assert!(matches!(c2.lock().unwrap().machine.current, End));
    }
}
```

1.3 Parser

```
use std::fs;
use std::path::Path;

use crate::channel::SessionAction;
use crate::error::ParserError::{self, *};
use crate::syntax::{CmlTerm, SessionTerm, Type, Value};

#[derive(Debug, Clone, PartialEq)]
enum Token {
    // Punctuation
    LParen,
    RParen,
    LBrace,
    RBrace,
    Lt,
    Gt,
```

```
    Dot,
    Comma,
    Semi,
    Pipe,
    PipePipe,
    Colon,
    // Operators
    Plus,
    Eq,
    EqEq,
    ThinArrow,
    FatArrow,
    Bang,
    Oplus,
    Amp,
    // Keywords
    FunKw,
    FixKw,
    IfKw,
    ThenKw,
    ElseKw,
    RefKw,
    GetKw,
    SetKw,
    NewKw,
    SendKw,
    RecvKw,
    SelectKw,
    OfferKw,
    LinkKw,
    EndKw,
    RecKw,
    // keywords (types)
    UnitTy,
    IntTy,
    BoolTy,
    ChannelTy,
    // atoms
    TrueKw,
    FalseKw,
    Ident(String),
    IntKw(i64),
}

struct Lexer<'a> {
    s: &'a [u8],
    i: usize,
    n: usize,
    line: usize,
    col: usize,
}

impl<'a> Lexer<'a> {
    fn new(src: &'a str) -> Self {
        Self {
            s: src.as_bytes(),
            i: 0,
            n: src.len(),
            line: 1,
            col: 1,
        }
    }

    fn peek(&self) -> Option<u8> {
        (self.i < self.n).then(|| self.s[self.i])
    }

    fn bump(&mut self) -> Option<u8> {
```

```
        let c = self.peek()?;
        self.i += 1;
        if c == b'\n' {
            self.line += 1;
            self.col = 1;
        } else {
            self.col += 1;
        }
        Some(c)
    }

    fn bump_many(&mut self, k: usize) {
        self.i += k;
        self.col += k;
    }

    fn eat_while<F: Fn(u8) -> bool>(&mut self, f: F) -> &'a [u8] {
        let start = self.i;
        while self.i < self.n && f(self.s[self.i]) {
            self.bump();
        }
        &self.s[start..self.i]
    }

    fn match_bytes(&mut self, pat: &[u8]) -> bool {
        if self.i + pat.len() ≤ self.n && &self.s[self.i..self.i + pat.len()] == pat {
            self.bump_many(pat.len());
            true
        } else {
            false
        }
    }

    fn skip_ws_comments(&mut self, file: &str) -> Result<(), ParserError> {
        loop {
            while let Some(c) = self.peek() {
                if matches!(c, b' ' | b'\t' | b'\r' | b'\n') {
                    self.bump();
                } else {
                    break;
                }
            }

            if self.match_bytes(b"//") {
                while let Some(c) = self.peek() {
                    self.bump();
                    if c == b'\n' {
                        break;
                    }
                }
                continue;
            }

            if self.match_bytes(b"/*") {
                let mut closed = false;
                while self.i < self.n {
                    if self.match_bytes(b"*/") {
                        closed = true;
                        break;
                    }
                    self.bump();
                }
                if !closed {
                    return Err(LexError {
                        file: file.into(),
                        msg: "Unterminated comment".into(),
                        line: self.line,
                        column: self.col,
```

```
                    });
                }
                continue;
            }
            break;
        }
        Ok(())
    }

    fn next_token(&mut self, file: &str) -> Result<Option<Cursor>, ParserError> {
        self.skip_ws_comments(file)?;
        let _c = match self.peek() {
            Some(c) => c,
            None => return Ok(None),
        };

        let (start_line, start_col) = (self.line, self.col);

        if self.match_bytes(b"==") {
            return Ok(Some(make(Token::EqEq, start_line, start_col)));
        }

        if self.match_bytes(b"⇒") {
            return Ok(Some(make(Token::FatArrow, start_line, start_col)));
        }

        if self.match_bytes(b"->") {
            return Ok(Some(make(Token::ThinArrow, start_line, start_col)));
        }

        if self.match_bytes(b"||") {
            return Ok(Some(make(Token::PipePipe, start_line, start_col)));
        }

        if self.match_bytes(b"&") {
            return Ok(Some(make(Token::Amp, start_line, start_col)));
        }

        if self.match_bytes("⊕".as_bytes()) {
            return Ok(Some(make(Token::Oplus, start_line, start_col)));
        }

        match self.bump().unwrap() {
            b'(' => Ok(Some(make(Token::LParen, start_line, start_col))),
            b')' => Ok(Some(make(Token::RParen, start_line, start_col))),
            b'{' => Ok(Some(make(Token::LBrace, start_line, start_col))),
            b'}' => Ok(Some(make(Token::RBrace, start_line, start_col))),
            b'<' => Ok(Some(make(Token::Lt, start_line, start_col))),
            b'>' => Ok(Some(make(Token::Gt, start_line, start_col))),
            b'.' => Ok(Some(make(Token::Dot, start_line, start_col))),
            b',' => Ok(Some(make(Token::Comma, start_line, start_col))),
            b';' => Ok(Some(make(Token::Semi, start_line, start_col))),
            b'|' => Ok(Some(make(Token::Pipe, start_line, start_col))),
            b':' => Ok(Some(make(Token::Colon, start_line, start_col))),
            b'+' => Ok(Some(make(Token::Plus, start_line, start_col))),
            b'!' => Ok(Some(make(Token::Bang, start_line, start_col))),
            b'=' => Ok(Some(make(Token::Eq, start_line, start_col))),
            d @ b'0'..=b'9' => {
                let mut v = (d - b'0') as i64;
                while let Some(dd @ b'0'..=b'9') = self.peek() {
                    self.bump();
                    v = v * 10 + (dd - b'0') as i64;
                }
                Ok(Some(make(Token::IntKw(v), start_line, start_col)))
            }

            ch if is_ident_start(ch) => {
                let rest = self.eat_while(is_ident_continue);
```

```
                let mut s = String::from_utf8(vec![ch]).unwrap();
                s.push_str(std::str::from_utf8(rest).unwrap());
                Ok(Some(make(keyword_or_ident(s), start_line, start_col)))
            }

            other => Err(LexError {
                file: file.into(),
                msg: format!("Invalid character: {}", other as char),
                line: start_line,
                column: start_col,
            }),
        }
    }

    fn tokenize(mut self, file: &str) -> Result<Vec<Cursor>, ParserError> {
        let mut tokens = Vec::new();
        while let Some(tok) = self.next_token(file)? {
            tokens.push(tok);
        }
        Ok(tokens)
    }
}

fn is_ident_start(char: u8) -> bool {
    char.is_ascii_lowercase() || char.is_ascii_uppercase() || char == b'_'
}

fn is_ident_continue(char: u8) -> bool {
    is_ident_start(char) || char.is_ascii_digit()
}

fn keyword_or_ident(string: String) -> Token {
    use Token::*;

    match string.as_str() {
        // terms
        "fun" => FunKw,
        "fix" => FixKw,
        "if" => IfKw,
        "then" => ThenKw,
        "else" => ElseKw,
        "ref" => RefKw,
        "get" => GetKw,
        "set" => SetKw,
        "new" => NewKw,
        "send" => SendKw,
        "recv" => RecvKw,
        "select" => SelectKw,
        "offer" => OfferKw,
        "link" => LinkKw,
        "end" => EndKw,
        "rec" => RecKw,
        "true" => TrueKw,
        "false" => FalseKw,
        // types
        "Unit" => UnitTy,
        "Int" => IntTy,
        "Bool" => BoolTy,
        "Chan" => ChannelTy,

        _ => Ident(string),
    }
}

struct Parser {
    tokens: Vec<Cursor>,
    pos: usize,
    file: String,
```

```rust
}

impl Parser {
    fn new_with_file(tokens: Vec<Cursor>, file: impl Into<String>) -> Self {
        Self {
            tokens,
            pos: 0,
            file: file.into(),
        }
    }

    fn here(&self) -> (usize, usize) {
        if let Some(cur) = self.peek() {
            (cur.line, cur.column)
        } else if let Some(last) = self.tokens.last() {
            (last.line, last.column + 1)
        } else {
            (1, 1)
        }
    }

    fn peek(&self) -> Option<&Cursor> {
        self.tokens.get(self.pos)
    }

    fn peek_kind(&self) -> Option<&Token> {
        self.tokens.get(self.pos).map(|c| &c.kind)
    }

    fn bump(&mut self) -> Option<Cursor> {
        if self.pos < self.tokens.len() {
            let cur = self.tokens[self.pos].clone();
            self.pos += 1;
            Some(cur)
        } else {
            None
        }
    }

    fn bump_kind(&mut self) -> Option<Token> {
        self.bump().map(|c| c.kind)
    }

    fn eat(&mut self, expected: &Token) -> Result<(), ParserError> {
        let got = self.bump().ok_or_else(|| {
            let (line, column) = self.here();
            UnexpectedEof {
                file: self.file.clone(),
                line,
                column,
            }
        })?;
        if &got.kind == expected {
            Ok(())
        } else {
            Err(UnexpectedToken {
                expected: format_token(expected),
                found: format_token(&got.kind),
                file: self.file.clone(),
                line: got.line,
                column: got.column,
            })
        }
    }

    fn eat_if(&mut self, f: impl FnOnce(&Token) -> bool) -> Option<Token> {
        if let Some(cur) = self.peek()
            && f(&cur.kind)
```

```
357             {
358                 return self.bump_kind();
359             }
360
361             None
362         }
363
364         fn expect_ident(&mut self, what: &'static str) -> Result<String, ParserError> {
365             match self.bump() {
366                 Some(cur) => match cur.kind {
367                     Token::Ident(s) => Ok(s),
368                     other => Err(UnexpectedToken {
369                         file: self.file.clone(),
370                         expected: "identifier".into(),
371                         found: format_token(&other),
372                         line: cur.line,
373                         column: cur.column,
374                     }),
375                 },
376                 None => {
377                     let (line, column) = self.here();
378                     Err(ExpectedIdent {
379                         file: self.file.clone(),
380                         what,
381                         line,
382                         column,
383                     })
384                 }
385             }
386         }
387
388         fn parse_program(&mut self) -> Result<SessionTerm, ParserError> {
389             let parsed = self.parse_session_term()?;
390             if let Some(cur) = self.peek() {
391                 return Err(TrailingInput {
392                     file: self.file.clone(),
393                     found: format_token(&cur.kind),
394                     line: cur.line,
395                     column: cur.column,
396                 });
397             }
398             Ok(parsed)
399         }
400
401         fn parse_session_term(&mut self) -> Result<SessionTerm, ParserError> {
402             self.parse_parallel()
403         }
404
405         fn parse_parallel(&mut self) -> Result<SessionTerm, ParserError> {
406             let mut left = self.parse_repl()?;
407             while matches!(self.peek_kind(), Some(Token::PipePipe)) {
408                 self.bump_kind();
409                 let right = self.parse_repl()?;
410                 left = SessionTerm::Parallel(Box::new(left), Box::new(right));
411             }
412
413             Ok(left)
414         }
415
416         fn parse_repl(&mut self) -> Result<SessionTerm, ParserError> {
417             if matches!(self.peek_kind(), Some(Token::Bang)) {
418                 self.bump_kind();
419                 let body = self.parse_repl()?;
420                 return Ok(SessionTerm::Replicate(Box::new(body)));
421             }
422             self.parse_session_primary()
423         }
424
```

```
425     fn parse_session_primary(&mut self) -> Result<SessionTerm, ParserError> {
426         use CmlTerm::*;
427         use SessionTerm::*;
428         use Token::*;
429         use Value::*;
430         match self.peek_kind() {
431             Some(NewKw) => {
432                 self.bump_kind();
433                 let chan_name = self.expect_ident("channel binding")?;
434                 self.eat(&Colon)?;
435                 let session_type = self.parse_action()?;
436                 self.eat(&Dot)?;
437                 let body = self.parse_session_term()?;
438                 Ok(NewChannel(chan_name, Some(session_type), Box::new(body)))
439             }
440 
441             Some(SendKw) => {
442                 self.bump_kind();
443                 let chan_name = self.parse_cml_term()?;
444                 if self.eat_if(|t| matches!(t, Comma)).is_none() {
445                     let cur = self.peek().unwrap();
446                     return Err(ExpectedComma {
447                         file: self.file.clone(),
448                         line: cur.line,
449                         column: cur.column,
450                     });
451                 }
452                 let value = self.parse_cml_term()?;
453 
454                 let cont = if self.eat_if(|t| matches!(t, Semi)).is_some() {
455                     self.parse_session_term()?
456                 } else {
457                     End
458                 };
459                 Ok(Send(chan_name, value, Box::new(cont)))
460             }
461 
462             Some(RecvKw) => {
463                 self.bump_kind();
464                 let chan_name = self.parse_cml_term()?;
465                 if self.eat_if(|t| matches!(t, Comma)).is_none() {
466                     let cur = self.peek().unwrap();
467                     return Err(ExpectedComma {
468                         file: self.file.clone(),
469                         line: cur.line,
470                         column: cur.column,
471                     });
472                 }
473                 let var_name = self.expect_ident("received variable")?;
474                 let cont = if self.eat_if(|t| matches!(t, Semi)).is_some() {
475                     self.parse_session_term()?
476                 } else {
477                     End
478                 };
479                 Ok(Receive(chan_name, var_name, Box::new(cont)))
480             }
481 
482             Some(SelectKw) => {
483                 self.bump_kind();
484                 let chan_name = self.parse_cml_term()?;
485                 if self.eat_if(|t| matches!(t, Comma)).is_none() {
486                     let cur = self.peek().unwrap();
487                     return Err(ExpectedComma {
488                         file: self.file.clone(),
489                         line: cur.line,
490                         column: cur.column,
491                     });
492                 }
```

```
                let lbl = self.expect_ident("label")?;
                let lbl_term = Val(Label(lbl));
                let cont = if self.eat_if(|t| matches!(t, Semi)).is_some() {
                    self.parse_session_term()?
                } else {
                    End
                };
                Ok(Select(chan_name, lbl_term, Box::new(cont)))
            }

            Some(OfferKw) => {
                self.bump_kind();
                let chan_name = self.parse_cml_term()?;
                self.eat(&LBrace)?;
                let mut arms: Vec<(CmlTerm, Box<SessionTerm>)> = Vec::new();
                loop {
                    let lbl = self.expect_ident("label")?;
                    self.eat(&FatArrow)?;
                    let body = self.parse_session_term()?;
                    arms.push((Val(Label(lbl)), Box::new(body)));
                    if self.eat_if(|t| matches!(t, Pipe)).is_some() {
                        continue;
                    }
                    break;
                }

                self.eat(&RBrace)?;
                if arms.is_empty() {
                    return Err(EmptyBranchSet {
                        file: self.file.clone(),
                        line: self.tokens.last().map_or(1, |c| c.line),
                        column: self.tokens.last().map_or(1, |c| c.column + 1),
                    });
                }

                Ok(Offer(chan_name, arms))
            }

            Some(RecKw) => {
                self.bump_kind();
                let var = self.expect_ident("recursion variable")?;
                self.eat(&Dot)?;
                let body = self.parse_session_term()?;
                Ok(Rec(var, Box::new(body)))
            }

            Some(Ident(_)) => {
                let x = if let Some(Ident(str)) = self.bump_kind() {
                    str
                } else {
                    unreachable!()
                };
                Ok(SVar(x))
            }

            Some(LinkKw) => {
                self.bump_kind();
                self.eat(&LParen)?;
                let chan1 = self.parse_cml_term()?;
                if self.eat_if(|t| matches!(t, Comma)).is_none() {
                    let cur = self.peek().unwrap();
                    return Err(ExpectedComma {
                        file: self.file.clone(),
                        line: cur.line,
                        column: cur.column,
                    });
                }
                let chan2 = self.parse_cml_term()?;
```

```
                self.eat(&RParen)?;
                let cont = if self.eat_if(|t| matches!(t, Semi)).is_some() {
                    self.parse_session_term()?
                } else {
                    End
                };
                Ok(Link(chan1, chan2, Box::new(cont)))
            }

            Some(EndKw) => {
                self.bump_kind();
                Ok(End)
            }

            Some(LParen) => {
                self.bump_kind();
                let inner = self.parse_session_term()?;
                self.eat(&RParen)?;
                Ok(inner)
            }

            Some(tok) => {
                let cur = self.peek().unwrap();
                Err(UnexpectedToken {
                    file: self.file.clone(),
                    expected: "session term".into(),
                    found: format_token(tok),
                    line: cur.line,
                    column: cur.column,
                })
            }

            None => {
                let (line, column) = self.here();
                Err(UnexpectedEof {
                    file: self.file.clone(),
                    line,
                    column,
                })
            }
        }
    }

    fn parse_cml_term(&mut self) -> Result<CmlTerm, ParserError> {
        self.parse_cml_if()
    }

    fn parse_cml_if(&mut self) -> Result<CmlTerm, ParserError> {
        if matches!(self.peek_kind(), Some(Token::IfKw)) {
            self.bump_kind();
            let cond = self.parse_cml_term()?;
            self.eat(&Token::ThenKw)?;
            let then_branch = self.parse_cml_term()?;
            self.eat(&Token::ElseKw)?;
            let else_branch = self.parse_cml_term()?;
            Ok(CmlTerm::If(
                Box::new(cond),
                Box::new(then_branch),
                Box::new(else_branch),
            ))
        } else {
            self.parse_cml_equal()
        }
    }

    fn parse_cml_equal(&mut self) -> Result<CmlTerm, ParserError> {
        let mut left = self.parse_cml_add()?;
        while matches!(self.peek_kind(), Some(Token::EqEq)) {
```

```
629             self.bump_kind();
630             let right = self.parse_cml_add()?;
631             left = CmlTerm::Equal(Box::new(left), Box::new(right));
632         }
633         Ok(left)
634     }
635
636     fn parse_cml_add(&mut self) -> Result<CmlTerm, ParserError> {
637         let mut left = self.parse_cml_app_or_annotation()?;
638         while matches!(self.peek_kind(), Some(Token::Plus)) {
639             self.bump_kind();
640             let right = self.parse_cml_app_or_annotation()?;
641             left = CmlTerm::Add(Box::new(left), Box::new(right));
642         }
643         Ok(left)
644     }
645
646     fn parse_cml_app_or_annotation(&mut self) -> Result<CmlTerm, ParserError> {
647         use Token::*;
648         let mut tok = self.parse_cml_atom()?;
649         if matches!(self.peek_kind(), Some(Colon)) {
650             self.bump_kind();
651             let ty = self.parse_type()?;
652             return Ok(CmlTerm::Annot(Box::new(tok), ty));
653         }
654
655         loop {
656             match self.peek_kind() {
657                 Some(RParen | RBrace | Semi | Comma | ThenKw | ElseKw | FatArrow | Colon) | None => {
658                     break;
659                 }
660
661                 Some(EqEq | Plus) => break,
662
663                 Some(
664                     SendKw | RecvKw | SelectKw | OfferKw | NewKw | RecKw | LinkKw | EndKw | Bang
665                     | Amp | Oplus,
666                 ) => break,
667
668                 _ => {}
669             }
670
671             let save = self.pos;
672             if let Ok(arg) = self.parse_cml_atom() {
673                 tok = CmlTerm::App(Box::new(tok), Box::new(arg));
674             } else {
675                 self.pos = save;
676                 break;
677             }
678         }
679
680         Ok(tok)
681     }
682
683     fn parse_cml_atom(&mut self) -> Result<CmlTerm, ParserError> {
684         use CmlTerm::*;
685         use Token::*;
686         use Value::*;
687         match self.peek_kind() {
688             Some(FunKw) => {
689                 self.bump_kind();
690                 let name = self.expect_ident("function name")?;
691                 self.eat(&LParen)?;
692                 let param = self.expect_ident("function parameter")?;
693                 self.eat(&Colon)?;
694                 let param_ty = self.parse_type()?;
695                 self.eat(&RParen)?;
696                 self.eat(&ThinArrow)?;
```

```
                let ret_ty = self.parse_type()?;
                self.eat(&LBrace)?;
                let body = self.parse_cml_term()?;
                self.eat(&RBrace)?;

                let fun_val = Val(Lambda {
                    name: Some(name),
                    param,
                    param_type: Some(param_ty.clone()),
                    body: Box::new(body)
                });

                Ok(Annot(Box::new(fun_val), Type::Function(Box::new(param_ty), Box::new(ret_ty))))
            }

            Some(FixKw) => {
                self.bump_kind();
                self.eat(&LParen)?;
                let func = self.expect_ident("fix binder")?;
                self.eat(&Colon)?;
                let dom_ty = self.parse_type()?;
                self.eat(&ThinArrow)?;
                let cod_ty = self.parse_type()?;
                self.eat(&RParen)?;
                self.eat(&FatArrow)?;
                let body = self.parse_cml_term()?;
                Ok(Fix(func, Some((dom_ty, cod_ty)), Box::new(body)))
            }

            Some(RefKw) => {
                self.bump_kind();
                let expr = self.parse_cml_term()?;
                Ok(Ref(Box::new(expr)))
            }

            Some(GetKw) => {
                self.bump_kind();
                let expr = self.parse_cml_term()?;
                Ok(Get(Box::new(expr)))
            }

            Some(SetKw) => {
                self.bump_kind();
                let expr1 = self.parse_cml_term()?;
                if self.eat_if(|t| matches!(t, Eq)).is_none() {
                    let cur = self.peek().unwrap();
                    return Err(ExpectedEqual {
                        file: self.file.clone(),
                        line: cur.line,
                        column: cur.column,
                    });
                }
                let expr2 = self.parse_cml_term()?;
                Ok(Set(Box::new(expr1), Box::new(expr2)))
            }

            Some(TrueKw) => {
                self.bump_kind();
                Ok(Val(Bool(true)))
            }

            Some(FalseKw) => {
                self.bump_kind();
                Ok(Val(Bool(false)))
            }

            Some(UnitTy) => {
                self.bump_kind();
```

```
765                 Ok(Val(Unit))
766             }
767
768             Some(IntKw(_)) => {
769                 let n = if let Some(IntKw(i)) = self.bump_kind() {
770                     i
771                 } else {
772                     unreachable!()
773                 };
774                 Ok(Val(Int(n)))
775             }
776
777             Some(Ident(_)) => {
778                 let x = if let Some(Ident(str)) = self.bump_kind() {
779                     str
780                 } else {
781                     unreachable!()
782                 };
783                 Ok(Val(Var(x)))
784             }
785
786             Some(LParen) => {
787                 self.bump_kind();
788                 let inner = self.parse_cml_term()?;
789                 self.eat(&RParen)?;
790                 Ok(inner)
791             }
792
793             Some(tok) => {
794                 let cur = self.peek().unwrap();
795                 Err(UnexpectedToken {
796                     file: self.file.clone(),
797                     expected: "CML term".into(),
798                     found: format_token(tok),
799                     line: cur.line,
800                     column: cur.column,
801                 })
802             }
803
804             None => {
805                 let (line, column) = self.here();
806                 Err(UnexpectedEof {
807                     file: self.file.clone(),
808                     line,
809                     column,
810                 })
811             }
812         }
813     }
814
815     fn parse_type(&mut self) -> Result<Type, ParserError> {
816         self.parse_type_arrow()
817     }
818
819     fn parse_type_arrow(&mut self) -> Result<Type, ParserError> {
820         let lhs = self.parse_type_atom()?;
821         if matches!(self.peek_kind(), Some(Token::FatArrow)) {
822             self.bump_kind();
823             let rhs = self.parse_type_atom()?;
824             Ok(Type::Function(Box::new(lhs), Box::new(rhs)))
825         } else {
826             Ok(lhs)
827         }
828     }
829
830     fn parse_type_atom(&mut self) -> Result<Type, ParserError> {
831         use Token::*;
832         use Type::*;
```

```
        match self.peek_kind() {
            Some(UnitTy) => {
                self.bump_kind();
                Ok(Unit)
            }

            Some(IntTy) => {
                self.bump_kind();
                Ok(Int)
            }

            Some(BoolTy) => {
                self.bump_kind();
                Ok(Bool)
            }

            Some(ChannelTy) => {
                self.bump_kind();
                let session_ty = self.parse_action()?;
                Ok(Channel(Box::new(Type::Session(Box::new(session_ty)))))
            }

            Some(LParen) => {
                self.bump_kind();
                let inner = self.parse_type()?;
                self.eat(&RParen)?;
                Ok(inner)
            }

            Some(tok) => {
                let cur = self.peek().unwrap();
                Err(UnexpectedToken {
                    file: self.file.clone(),
                    expected: "type".into(),
                    found: format_token(tok),
                    line: cur.line,
                    column: cur.column,
                })
            }

            None => {
                let (line, column) = self.here();
                Err(UnexpectedEof {
                    file: self.file.clone(),
                    line,
                    column,
                })
            }
        }
    }

    fn parse_action(&mut self) -> Result<SessionAction, ParserError> {
        use SessionAction::*;
        use Token::*;
        match self.peek_kind() {
            Some(EndKw) => {
                self.bump_kind();
                Ok(End)
            }

            Some(SendKw) => {
                self.bump_kind();
                if self.eat_if(|t| matches!(t, Lt)).is_none() {
                    let cur = self.peek().unwrap();
                    return Err(ExpectedCommType {
                        file: self.file.clone(),
                        line: cur.line,
                        column: cur.column,
```

```
                    });
                }
                let ty = self.parse_type()?;
                self.eat(&Gt)?;
                self.eat(&Semi)?;
                let cont = self.parse_action()?;
                Ok(SendAction(ty, Box::new(cont)))
            }

            Some(RecvKw) => {
                self.bump_kind();
                if self.eat_if(|t| matches!(t, Lt)).is_none() {
                    let cur = self.peek().unwrap();
                    return Err(ExpectedCommType {
                        file: self.file.clone(),
                        line: cur.line,
                        column: cur.column,
                    });
                }
                let ty = self.parse_type()?;
                self.eat(&Gt)?;
                self.eat(&Semi)?;
                let cont = self.parse_action()?;
                Ok(ReceiveAction(ty, Box::new(cont)))
            }

            Some(SelectKw) | Some(Oplus) => {
                self.bump_kind();
                self.eat(&LBrace)?;
                let mut arms: Vec<(String, SessionAction)> = Vec::new();
                loop {
                    let lbl = self.expect_ident("label")?;
                    self.eat(&Colon)?;
                    let body = self.parse_action()?;
                    if arms.iter().any(|(k, _)| k == &lbl) {
                        return Err(DuplicateLabel {
                            file: self.file.clone(),
                            label: lbl,
                            line: self.tokens.last().map_or(1, |c| c.line),
                            column: self.tokens.last().map_or(1, |c| c.column + 1),
                        });
                    }
                    arms.push((lbl, body));
                    if self.eat_if(|t| matches!(t, Pipe)).is_some() {
                        continue;
                    }
                    break;
                }
                self.eat(&RBrace)?;
                if arms.is_empty() {
                    return Err(EmptyBranchSet {
                        file: self.file.clone(),
                        line: self.tokens.last().map_or(1, |c| c.line),
                        column: self.tokens.last().map_or(1, |c| c.column + 1),
                    });
                }
                Ok(SelectAction(arms))
            }

            Some(OfferKw) | Some(Amp) => {
                self.bump_kind();
                self.eat(&LBrace)?;
                let mut arms: Vec<(String, SessionAction)> = Vec::new();
                loop {
                    let lbl = self.expect_ident("label")?;
                    self.eat(&Colon)?;
                    let body = self.parse_action()?;
                    if arms.iter().any(|(k, _)| k == &lbl) {
```

```
                        let (line, column) = self.here();
                        return Err(DuplicateLabel {
                            file: self.file.clone(),
                            label: lbl,
                            line,
                            column,
                        });
                    }
                    arms.push((lbl, body));
                    if self.eat_if(|t| matches!(t, Pipe)).is_some() {
                        continue;
                    }
                    break;
                }

                self.eat(&RBrace)?;
                if arms.is_empty() {
                    let (line, column) = self.here();
                    return Err(EmptyBranchSet {
                        file: self.file.clone(),
                        line,
                        column,
                    });
                }
                Ok(BranchAction(arms))
            }

            Some(RecKw) => {
                self.bump_kind();
                let var = self.expect_ident("recursion variable")?;
                self.eat(&Dot)?;
                let body = self.parse_action()?;
                Ok(RecAction(var, Box::new(body)))
            }

            Some(Ident(_)) => {
                let var = if let Some(Ident(str)) = self.bump_kind() {
                    str
                } else {
                    unreachable!()
                };
                Ok(Var(var))
            }

            Some(LParen) => {
                self.bump_kind();
                let inner = self.parse_action()?;
                self.eat(&RParen)?;
                Ok(inner)
            }

            Some(tok) => {
                let cur = self.peek().unwrap();
                Err(UnexpectedToken {
                    file: self.file.clone(),
                    expected: "session type".into(),
                    found: format_token(tok),
                    line: cur.line,
                    column: cur.column,
                })
            }

            None => {
                let (line, column) = self.here();
                Err(UnexpectedEof {
                    file: self.file.clone(),
                    line,
                    column,
```

```
                })
            }
        }
    }
}

fn format_token(tok: &Token) -> String {
    use Token::*;
    match tok {
        LParen => "'('".into(),
        RParen => "')'".into(),
        LBrace => "'{'".into(),
        RBrace => "'}'".into(),
        Lt => "'<'".into(),
        Gt => "'>'".into(),
        Dot => "'.'".into(),
        Comma => "','".into(),
        Semi => "';'".into(),
        Pipe => "'|'".into(),
        PipePipe => "'||'".into(),
        Colon => "':'".into(),
        Plus => "'+'".into(),
        Eq => "'='".into(),
        EqEq => "'=='".into(),
        FatArrow => "'⇒'".into(),
        ThinArrow => "'->'".into(),
        Bang => "'!'".into(),
        FunKw => "'fun'".into(),
        FixKw => "'fix'".into(),
        IfKw => "'if'".into(),
        ThenKw => "'then'".into(),
        ElseKw => "'else'".into(),
        RefKw => "'ref'".into(),
        GetKw => "'get'".into(),
        SetKw => "'set'".into(),
        NewKw => "'new'".into(),
        SendKw => "'send'".into(),
        RecvKw => "'recv'".into(),
        SelectKw => "'select'".into(),
        OfferKw => "'offer'".into(),
        LinkKw => "'link'".into(),
        EndKw => "'end'".into(),
        RecKw => "'rec'".into(),
        UnitTy => "'unit'".into(),
        IntTy => "'int'".into(),
        BoolTy => "'bool'".into(),
        ChannelTy => "'channel'".into(),
        TrueKw => "'true'".into(),
        FalseKw => "'false'".into(),
        Ident(_) => "identifier".into(),
        IntKw(_) => "integer".into(),
        Oplus => "'sel'".into(),
        Amp => "'&'".into(),
    }
}

#[derive(Clone, Debug, PartialEq)]
struct Cursor {
    kind: Token,
    line: usize,
    column: usize,
}

fn make(tok: Token, line: usize, column: usize) -> Cursor {
    Cursor {
        kind: tok,
        line,
        column,
```

```
    }
}

pub fn parse_program_in(file: &str, src: &str) -> Result<SessionTerm, ParserError> {
    let tokens = Lexer::new(src).tokenize(file)?;
    let mut parser = Parser::new_with_file(tokens, file);
    parser.parse_program()
}

pub fn parse_type_in(file: &str, src: &str) -> Result<Type, ParserError> {
    let tokens = Lexer::new(src).tokenize(file)?;
    let mut parser = Parser::new_with_file(tokens, file);
    parser.parse_type()
}

pub fn parse_program(src: &str) -> Result<SessionTerm, ParserError> {
    parse_program_in("<stdin>", src)
}

pub fn parse_type_src(src: &str) -> Result<Type, ParserError> {
    parse_type_in("<stdin>", src)
}

pub fn parse_program_file<P: AsRef<Path>>(path: P) -> Result<SessionTerm, ParserError> {
    let p = path.as_ref();
    let src = fs::read_to_string(p).map_err(|e| ParserError::IoError {
        file: p.to_string_lossy().into_owned(),
        msg: e.to_string(),
    })?;
    parse_program_in(&p.to_string_lossy(), &src)
}

pub fn parse_type_file<P: AsRef<Path>>(path: P) -> Result<Type, ParserError> {
    let p = path.as_ref();
    let src = fs::read_to_string(p).map_err(|e| ParserError::IoError {
        file: p.to_string_lossy().into_owned(),
        msg: e.to_string(),
    })?;
    parse_type_in(&p.to_string_lossy(), &src)
}
```

1.4 Evaluation

```
use crate::channel::ChannelInternal;
use crate::error::Anyhow;
use crate::error::{
    ArenaError::{self, *},
    TypeError::*,
    SessionError::*,
};
use crate::syntax::{
    CmlTerm, Env, Memory, SessionTerm, Value, insert_at_new_scope, insert_var, lookup_var,
};
use std::collections::HashMap;
use std::error::Error;
use std::sync::{Arc, Mutex};
use std::thread;

pub fn eval_pure(
    term: &CmlTerm,
    env: &Env,
    memory: &mut Memory,
) -> Result<Value, Box<dyn Error + Sync + Send>> {
    use CmlTerm::*;
    use Value::*;
    match term {
```

```
24          Val(v) ⇒ match v {
25              Var(name) ⇒ {
26                  let var = lookup_var(env, name);
27                  if let Some(var) = var {
28                      Ok(var)
29                  } else {
30                      Err(Box::new(UndefinedVariable(name.clone())))
31                  }
32              }
33              Lambda {
34                  name: self_name,
35                  param,
36                  param_type,
37                  body,
38              } ⇒ Ok(Lambda {
39                  name: self_name.clone(),
40                  param: param.clone(),
41                  param_type: param_type.clone(),
42                  body: body.clone(),
43              }),
44              Int(i) ⇒ Ok(Int(*i)),
45              Bool(b) ⇒ Ok(Bool(*b)),
46              Channel(internal) ⇒ Ok(Channel(internal.clone())),
47              Unit ⇒ Ok(Unit),
48              Address(addr) ⇒ Ok(Address(*addr)),
49              Label(label) ⇒ Ok(Label(label.clone())),
50          },
51
52          Annot(inner, _ty) ⇒ eval_pure(inner, env, memory),
53
54          App(f, arg) ⇒ {
55              let f_val = eval_pure(f, env, memory)?;
56              let arg_val = eval_pure(arg, env, memory)?;
57              if let Lambda { name, param, body, .. } = f_val {
58                  let mut new_env = env.clone();
59                  if let Some(self_name) = name {
60                      let self_val = Lambda {
61                          name: Some(self_name.clone()),
62                          param: param.clone(),
63                          param_type: None,
64                          body: body.clone(),
65                      };
66                      insert_at_new_scope(&mut new_env, self_name, self_val);
67                  } else {
68                      insert_at_new_scope(&mut new_env, param.clone(), arg_val.clone());
69                  }
70                  insert_var(&mut new_env, param.clone(), arg_val);
71                  eval_pure(&body, &new_env, memory)
72              } else {
73                  Err(Box::new(ExpectedFunction))
74              }
75          }
76
77          Fix(f, _annot, body) ⇒ match body.as_ref() {
78              Val(Lambda{param, param_type, body, ..}) ⇒ Ok(Lambda {
79                  name: Some(f.clone()),
80                  param: param.clone(),
81                  param_type:  param_type.clone(),
82                  body: body.clone(),
83              }),
84              _ ⇒ Err(Box::new(ExpectedRecursiveLambda)),
85          },
86
87          If(cond, t, f) ⇒ {
88              let cond_val = eval_pure(cond, env, memory)?;
89              match cond_val {
90                  Bool(true) ⇒ eval_pure(t, env, memory),
91                  Bool(false) ⇒ eval_pure(f, env, memory),
```

```
                _ => Err(Box::new(ExpectedBoolean)),
            }
        }

        Add(lhs, rhs) => {
            let lhs_val = eval_pure(lhs, env, memory)?;
            let rhs_val = eval_pure(rhs, env, memory)?;
            match (lhs_val, rhs_val) {
                (Int(l), Int(r)) => Ok(Int(l + r)),
                _ => Err(Box::new(ExpectedInteger)),
            }
        }

        Equal(lhs, rhs) => {
            let lhs_val = eval_pure(lhs, env, memory)?;
            let rhs_val = eval_pure(rhs, env, memory)?;
            Ok(Bool(lhs_val == rhs_val))
        }

        Ref(t) => {
            let t_val = eval_pure(t, env, memory)?;
            let addr = memory.alloc(t_val);
            Ok(Address(addr))
        }

        Get(t) => {
            let addr_val = eval_pure(t, env, memory)?;
            if let Address(addr) = addr_val {
                let value = memory.get(addr)?.clone();
                Ok(value)
            } else {
                Err(Box::new(ExpectedReference))
            }
        }

        Set(t1, t2) => {
            let addr_val = eval_pure(t1, env, memory)?;
            if let Address(addr) = addr_val {
                let value = eval_pure(t2, env, memory)?;
                memory.set(addr, value.clone())?;
                Ok(value)
            } else {
                Err(Box::new(ExpectedReference))
            }
        }

        Bottom => Ok(Unit),
    }
}

#[derive(Debug, Clone, PartialEq)]
pub enum Player {
    Proponent,
    Opponent,
}

#[derive(Debug, Clone, PartialEq)]
pub enum Event {
    NewChannel(String),
    Send(String),
    Receive(String),
    Select(String),
    Offer(String),
    Link(String, String),
    RecUnfold,
    Join,
    End,
}
```

```rust

impl From<Event> for String {
    fn from(val: Event) -> Self {
        use Event::*;
        match val {
            NewChannel(name) => format!("new_channel@{}", name),
            Send(name) => format!("send@{}", name),
            Receive(name) => format!("receive@{}", name),
            Select(name) => format!("select@{}", name),
            Offer(name) => format!("offer@{}", name),
            Link(c1, c2) => format!("link@{}-{}", c1, c2),
            RecUnfold => "rec_unfold".to_string(),
            Join => "join".to_string(),
            End => "end".to_string(),
        }
    }
}

impl std::fmt::Display for Event {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let moves = String::from(self.clone());
        write!(f, "{}", moves)
    }
}

pub type Events = Vec<Event>;

#[derive(Debug, Clone)]
pub struct DAGNode {
    pub id: usize,
    pub event: Event,
    pub player: Player,
    pub justification: Option<usize>,
    pub children: Vec<usize>,
}

impl DAGNode {
    pub fn new(id: usize, event: Event, player: Player, justification: Option<usize>) -> Self {
        DAGNode {
            id,
            event,
            player,
            justification,
            children: Vec::new(),
        }
    }
}

#[derive(Debug, Clone, Default)]
pub struct Arena {
    pub nodes: Vec<DAGNode>,
    pub edges: Vec<(usize, usize)>,
}

impl Arena {
    pub fn new() -> Self {
        Arena {
            nodes: Vec::new(),
            edges: Vec::new(),
        }
    }

    pub fn add_node(&mut self, event: Event, player: Player, parent: Option<usize>) -> usize {
        let id = self.nodes.len();
        let node = DAGNode::new(id, event.clone(), player.clone(), parent);
        self.nodes.push(node);
        if let Some(parent_id) = parent {
            self.edges.push((parent_id, id));
```

```
            self.nodes[parent_id].children.push(id);
        }
        id
    }

    pub fn to_dot(&self) -> String {
        let mut dot = String::from("digraph Arena {\n");
        for node in &self.nodes {
            let label = format!("{:?} ({:?})", node.event, node.player);
            dot.push_str(&format!("    {} [label=\"{}\"];\n", node.id, label));
        }
        for (from, to) in &self.edges {
            dot.push_str(&format!("    {} -> {};\n", from, to));
        }
        dot.push_str("}\n");
        dot
    }

    pub fn validate_justification(
        &self,
        node_id: usize,
        parent_id: Option<usize>,
    ) -> Result<(), ArenaError> {
        let node = self.nodes.get(node_id).ok_or(InvalidNode(node_id))?;

        match (node.justification, parent_id) {
            (Some(expected), Some(actual)) if expected == actual => Ok(()),
            (None, None) => Ok(()),
            _ => Err(InvalidJustification),
        }?;

        if let Some(parent_id) = parent_id {
            let parent = self.nodes.get(parent_id).ok_or(InvalidNode(parent_id))?;

            let _ = match (&parent.player, &node.player) {
                (Player::Proponent, Player::Opponent) | (Player::Opponent, Player::Proponent) => {
                    Ok(())
                }
                _ => Err(InvalidPlayerSequence),
            };
        }

        if let Some(parent_id) = parent_id {
            let parent = &self.nodes[parent_id];
            if !parent.children.contains(&node_id) {
                return Err(OrphanNode(node_id));
            }
        }

        Ok(())
    }
}

pub type SharedArena = Arc<Mutex<Arena>>;

#[inline]
fn add_node_sync(
    arena: &SharedArena,
    event: Event,
    player: Player,
    parent: Option<usize>,
) -> usize {
    let mut ar = arena.lock().unwrap();
    ar.add_node(event, player, parent)
}

#[derive(Debug, Clone)]
pub struct Process {
```

```rust
    pub term: SessionTerm,
    pub env: Env,
    pub memory: Memory,
    pub senv: HashMap<String, SessionTerm>,
    pub current_node: Option<usize>,
    pub causal_context: Vec<usize>,
    parallel_anchor: Option<usize>,
}

impl Process {
    pub fn new(term: SessionTerm) -> Self {
        Process {
            term,
            env: Env::new(),
            memory: Memory::new(),
            senv: HashMap::new(),
            current_node: None,
            causal_context: Vec::new(),
            parallel_anchor: None,
        }
    }

    pub fn with_causal_node(&self, node_id: usize) -> Self {
        let mut new_process = self.clone();
        new_process.causal_context.push(node_id);
        new_process.current_node = Some(node_id);
        new_process
    }

    pub fn with_term(&self, term: SessionTerm) -> Self {
        let mut new_process = self.clone();
        new_process.term = term;
        new_process
    }

    pub fn fork_for_parallel(&self, term: SessionTerm, parent: Option<usize>) -> Self {
        let mut process = Process {
            term,
            env: self.env.clone(),
            memory: self.memory.clone(),
            senv: self.senv.clone(),
            current_node: None,
            causal_context: Vec::new(),
            parallel_anchor: parent,
        };
        if let Some(p) = parent {
            process.causal_context.push(p);
        }
        process
    }

    #[inline]
    pub fn anchor_parent(&self) -> Option<usize> {
        self.parallel_anchor
    }

    pub fn get_arena_nodes(&self) -> Option<(usize, usize)> {
        match (self.current_node, self.causal_context.last()) {
            (Some(current), Some(parent)) => Some((current, *parent)),
            _ => None,
        }
    }
}

pub fn eval_game(
    process: &mut Process,
    arena: &SharedArena,
) -> Result<Value, Box<dyn Error + Send + Sync>> {
```

```
364     let term = process.term.clone();
365     let memory = &mut process.memory;
366     match term {
367         SessionTerm::NewChannel(name, opt_proto, body) => {
368             let fresh_name = name.to_string();
369             let parent = process.causal_context.last().copied();
370             let node_id = add_node_sync(
371                 arena,
372                 Event::NewChannel(fresh_name.clone()),
373                 Player::Proponent,
374                 parent,
375             );
376             process.current_node = Some(node_id);
377 
378             let internal = match opt_proto.as_ref() {
379                 Some(s) => ChannelInternal::restricted_with_action(&fresh_name, s.clone(), node_id),
380                 None => ChannelInternal::restricted(&fresh_name, node_id),
381             };
382             let channel = Value::Channel(Arc::new(Mutex::new(internal)));
383 
384             let mut next = process.with_causal_node(node_id);
385             insert_var(&mut next.env, name.clone(), channel.clone());
386             next.term = *body;
387             eval_game(&mut next, arena)
388         }
389 
390         SessionTerm::Send(channel_term, message_term, cont) => {
391             let channel_val = eval_pure(&channel_term, &process.env, memory)?;
392             let message_val = eval_pure(&message_term, &process.env, memory)?;
393 
394             if let Value::Channel(arc_channel) = channel_val {
395                 let ch_name = { arc_channel.lock().unwrap().name.clone() };
396                 let parent = process.causal_context.last().copied();
397                 let node_id = add_node_sync(arena, Event::Send(ch_name), Player::Proponent, parent);
398 
399                 ChannelInternal::send(&arc_channel, message_val)?;
400 
401                 {
402                     let mut c = arc_channel.lock().unwrap();
403                     if let Some(peer_w) = &c.linked_to {
404                         if let Some(peer) = peer_w.upgrade() {
405                             peer.lock().unwrap().last_send_node = Some(node_id);
406                         } else {
407                             c.last_send_node = Some(node_id);
408                         }
409                     } else {
410                         c.last_send_node = Some(node_id);
411                     }
412                 }
413 
414                 let mut next = process.with_causal_node(node_id);
415                 next.current_node = Some(node_id);
416 
417                 next.term = *cont;
418                 return eval_game(&mut next, arena);
419             }
420 
421             Err(Box::new(ExpectedChannel))
422         }
423 
424         SessionTerm::Receive(channel_term, var, cont) => {
425             let channel_val = eval_pure(&channel_term, &process.env, memory)?;
426 
427             if let Value::Channel(arc_channel) = channel_val {
428                 let value = ChannelInternal::receive(&arc_channel)?;
429 
430                 let (ch_name, parent_send) = {
431                     let guard = arc_channel.lock().unwrap();
```

```
                    (guard.name.clone(), guard.last_send_node)
                };

                let send_node = parent_send.ok_or_else(|| Box::new(MissingCausalParent))?;

                let node_id = add_node_sync(
                    arena,
                    Event::Receive(ch_name),
                    Player::Opponent,
                    Some(send_node),
                );

                {
                    arc_channel.lock().unwrap().last_receive_node = Some(node_id);
                }

                let mut next = process.with_causal_node(node_id);
                next.current_node = Some(node_id);

                insert_var(&mut next.env, var.clone(), value);
                next.term = *cont;
                return eval_game(&mut next, arena);
            }

            Err(Box::new(ExpectedChannel))
        }

        SessionTerm::Select(channel_term, label_term, cont) => {
            let channel_val = eval_pure(&channel_term, &process.env, memory)?;
            let label_val = eval_pure(&label_term, &process.env, memory)?;

            if let Value::Channel(arc_channel) = channel_val {
                let ch_name = { arc_channel.lock().unwrap().name.clone() };
                let parent = process.anchor_parent();
                let node_id =
                    add_node_sync(arena, Event::Select(ch_name), Player::Proponent, parent);
                let mut next = process.with_causal_node(node_id);

                ChannelInternal::handle_select(&arc_channel, label_val, *cont, &mut next)?;
                next.current_node = Some(node_id);

                return eval_game(&mut next, arena);
            }

            Err(Box::new(ExpectedChannel))
        }

        SessionTerm::Offer(channel_term, branches) => {
            let channel_val = eval_pure(&channel_term, &process.env, memory)?;

            if let Value::Channel(arc_channel) = channel_val {
                let ch_name = { arc_channel.lock().unwrap().name.clone() };
                let parent = process.anchor_parent();
                let node_id = add_node_sync(arena, Event::Offer(ch_name), Player::Opponent, parent);
                let mut next = process.with_causal_node(node_id);

                let mut payload_branches = Vec::with_capacity(branches.len());
                for (label, body) in branches {
                    let memory = &mut next.memory;
                    let label_val = eval_pure(&label, &next.env, memory)?;
                    payload_branches.push((label_val, *body));
                }

                let _chosen =
                    ChannelInternal::handle_offer(&arc_channel, payload_branches, &mut next)?;

                next.current_node = Some(node_id);
                return eval_game(&mut next, arena);
```

```
500             }
501
502             Err(Box::new(ExpectedChannel))
503         }
504
505         SessionTerm::Link(chan1_term, chan2_term, cont) => {
506             let c1 = match eval_pure(&chan1_term, &process.env, memory)? {
507                 Value::Channel(i) => i,
508                 _ => return Err(Box::new(ExpectedChannel)),
509             };
510             let c2 = match eval_pure(&chan2_term, &process.env, memory)? {
511                 Value::Channel(i) => i,
512                 _ => return Err(Box::new(ExpectedChannel)),
513             };
514
515             let (name1, name2) = {
516                 let guard1 = c1.lock().unwrap();
517                 let guard2 = c2.lock().unwrap();
518                 (guard1.name.clone(), guard2.name.clone())
519             };
520
521             let parent = process.causal_context.last().copied();
522             let node_id =
523                 add_node_sync(arena, Event::Link(name1, name2), Player::Proponent, parent);
524             process.causal_context.push(node_id);
525             process.parallel_anchor = Some(node_id);
526             process.current_node = Some(node_id);
527
528             ChannelInternal::link_channels(&c1, &c2)?;
529             process.term = *cont;
530             eval_game(process, arena)
531         }
532
533         SessionTerm::SVar(x) => {
534             if let Some(t) = process.senv.get(&x).cloned() {
535                 process.term = t;
536                 eval_game(process, arena)
537             } else {
538                 Err(Box::new(UndefinedSessionVar(x)))
539             }
540         }
541
542         SessionTerm::Rec(var, body) => {
543             let rec_node = add_node_sync(
544                 arena,
545                 Event::RecUnfold,
546                 Player::Proponent,
547                 process.causal_context.last().copied(),
548             );
549             let mut next = process.with_causal_node(rec_node);
550             let rec_term = SessionTerm::Rec(var.clone(), body.clone());
551             next.senv.insert(var.clone(), rec_term);
552             next.current_node = Some(rec_node);
553
554             next.term = *body;
555             eval_game(&mut next, arena)
556         }
557
558         SessionTerm::Parallel(p1, p2) => {
559             let parent = process.anchor_parent();
560
561             let mut p1p = process.fork_for_parallel(*p1, parent);
562             let mut p2p = process.fork_for_parallel(*p2, parent);
563
564             let arena1 = Arc::clone(arena);
565             let arena2 = Arc::clone(arena);
566
567             thread::scope(|s| {
```

```
                let h1 = s.spawn(move || eval_game(&mut p1p, &arena1));
                let h2 = s.spawn(move || eval_game(&mut p2p, &arena2));

                let _r1 = h1
                    .join()
                    .map_err(|_| Anyhow("parallel thread 1 panicked"))??;
                let _r2 = h2
                    .join()
                    .map_err(|_| Anyhow("parallel thread 2 panicked"))??;

                add_node_sync(arena, Event::Join, Player::Proponent, parent);
                Ok::<_, Box<dyn std::error::Error + Send + Sync>>(Value::Unit)
            })
        }

        SessionTerm::Replicate(proc) ⇒ {
            let parent = process.causal_context.last().copied();

            let mut p1 = process.fork_for_parallel((*proc).clone(), parent);
            let mut p2 = process.fork_for_parallel((*proc).clone(), parent);

            let a1 = std::sync::Arc::clone(arena);
            let a2 = std::sync::Arc::clone(arena);

            std::thread::scope(|s| {
                let h1 = s.spawn(move || eval_game(&mut p1, &a1));
                let h2 = s.spawn(move || eval_game(&mut p2, &a2));

                let _v1 = h1
                    .join()
                    .map_err(|_| Anyhow("replicate thread 1 panicked"))??;
                let _v2 = h2
                    .join()
                    .map_err(|_| Anyhow("replicate thread 2 panicked"))??;

                add_node_sync(arena, Event::Join, Player::Proponent, parent);
                Ok::<_, Box<dyn std::error::Error + Send + Sync>>(Value::Unit)
            })
        }

        SessionTerm::End ⇒ {
            let parent = process.current_node;
            add_node_sync(
                arena,
                Event::End,
                Player::Proponent,
                parent
            );
            Ok(Value::Unit)
        }
    }
}
```

1.5 Runtime

```
use crate::channel::SessionAction;
use crate::error::{SessionError::*, TypeError::*};
use crate::game_semantics::{eval_game, Arena, Process, SharedArena};
use crate::syntax::{CmlTerm, SessionTerm, Type, Value};
use std::collections::HashMap;
use std::error::Error;
use std::sync::{Arc, Mutex};

#[derive(Debug, Clone, Default)]
pub struct TypeChecker {
    gamma: Vec<HashMap<String, Type>>,
```

```
    sigma: Vec<HashMap<String>>, // For session types
}

impl TypeChecker {
    pub fn new() -> Self {
        Self {
            gamma: vec![HashMap::new()],
            sigma: vec![HashMap::new()],
        }
    }

    fn gamma_push(&mut self) {
        self.gamma.push(HashMap::new());
    }

    fn gamma_pop(&mut self) {
        self.gamma.pop();
    }

    fn gamma_insert(&mut self, var: String, ty: Type) {
        self.gamma.last_mut().unwrap().insert(var, ty);
    }

    fn gamma_lookup(&self, x: &str) -> Option<Type> {
        for scope in self.gamma.iter().rev() {
            if let Some(ty) = scope.get(x) {
                return Some(ty.clone());
            }
        }
        None
    }

    fn sigma_push(&mut self) {
        self.sigma.push(HashMap::new());
    }

    fn sigma_pop(&mut self) {
        self.sigma.pop();
    }

    fn sigma_insert(&mut self, var: String, ty: Type) {
        self.sigma.last_mut().unwrap().insert(var, ty);
    }

    fn sigma_lookup(&self, x: &str) -> Option<Type> {
        for scope in self.sigma.iter().rev() {
            if let Some(ty) = scope.get(x) {
                return Some(ty.clone());
            }
        }
        None
    }

    pub fn cml(&mut self, term: &CmlTerm) -> Result<Type, Box<dyn Error>> {
        use CmlTerm::*;
        use Type::*;
        match term {
            Val(v) => match v {
                Value::Var(x) => self
                    .gamma_lookup(x)
                    .ok_or_else(|| Box::new(UndefinedVariable(x.clone())) as _),
                Value::Int(_) => Ok(Int),
                Value::Bool(_) => Ok(Bool),
                Value::Unit => Ok(Unit),
                Value::Channel(_) => Err(Box::new(ExpectedChannelType)),
                Value::Address(_) => Err(Box::new(AddressIsRuntimeOnly)),
                Value::Label(_) => Ok(Unit),
                Value::Lambda {
```

```
            name: _,
            param,
            param_type,
            body,
        } ⇒ {
            let param_ty = param_type
                .clone()
                .ok_or_else(|| Box::new(MissingTypeAnnotation))?;
            self.gamma_push();
            self.gamma_insert(param.clone(), param_ty.clone());
            let body_ty = self.cml(body)?;
            self.gamma_pop();
            Ok(Function(Box::new(param_ty), Box::new(body_ty)))
        }
    },

    App(f, a) ⇒ {
        let f_type = self.cml(f)?;
        let a_type = self.cml(a)?;
        if let Function(param_type, return_type) = f_type {
            if *param_type == a_type {
                Ok(*return_type)
            } else {
                Err(Box::new(FunctionTypeMismatch(*param_type, a_type)))
            }
        } else {
            Err(Box::new(ExpectedFunctionType))
        }
    }

    Fix(f, annot, body) ⇒ {
        let (dom, cod) = annot
            .clone()
            .ok_or_else(|| Box::new(MissingTypeAnnotation))?;
        self.gamma_push();
        self.gamma_insert(
            f.clone(),
            Function(Box::new(dom.clone()), Box::new(cod.clone())),
        );
        let body_type = self.cml(body)?;
        self.gamma_pop();
        if body_type ≠ cod {
            return Err(Box::new(TypeMismatch(body_type, cod.clone())));
        }
        Ok(Type::Function(Box::new(dom), Box::new(cod)))
    }

    If(cond, t, f) ⇒ {
        let cond_type = self.cml(cond)?;
        if cond_type ≠ Bool {
            return Err(Box::new(ExpectedBoolean));
        }
        let t_type = self.cml(t)?;
        let f_type = self.cml(f)?;
        if t_type ≠ f_type {
            return Err(Box::new(BranchTypeMismatch(t_type, f_type)));
        }
        Ok(t_type)
    }

    Add(l, r) ⇒ {
        let lt = self.cml(l)?;
        let rt = self.cml(r)?;
        if lt == Int && rt == Int {
            Ok(Int)
        } else {
            Err(Box::new(ExpectedInteger))
        }
```

```
148             }
149
150             Equal(l, r) ⇒ {
151                 let lt = self.cml(l)?;
152                 let rt = self.cml(r)?;
153                 if lt == Int && rt == Int {
154                     Ok(Bool)
155                 } else {
156                     Err(Box::new(ExpectedInteger))
157                 }
158             }
159
160             Ref(expr) ⇒ {
161                 let expr_type = self.cml(expr)?;
162                 Ok(RefType(Box::new(expr_type)))
163             }
164
165             Get(expr) ⇒ match self.cml(expr)? {
166                 RefType(inner_type) ⇒ Ok(*inner_type),
167                 _ ⇒ Err(Box::new(ExpectedReferenceType)),
168             },
169
170             Set(ref_expr, val_expr) ⇒ match self.cml(ref_expr)? {
171                 RefType(inner) ⇒ {
172                     let val_type = self.cml(val_expr)?;
173                     if val_type ≠ *inner {
174                         return Err(Box::new(TypeMismatch(val_type, *inner)));
175                     }
176                     Ok(Unit)
177                 }
178                 _ ⇒ Err(Box::new(ExpectedReferenceType)),
179             },
180
181             Annot(expr, ty) ⇒ {
182                 let expr_type = self.cml(expr)?;
183                 if &expr_type ≠ ty {
184                     return Err(Box::new(TypeMismatch(expr_type, ty.clone())));
185                 }
186                 Ok(ty.clone())
187             }
188
189             Bottom ⇒ Ok(Unit),
190         }
191     }
192
193     pub fn session(&mut self, term: &SessionTerm) -> Result<Type, Box<dyn Error>> {
194         use SessionTerm::*;
195         use Type::*;
196         match term {
197             NewChannel(name, Some(s), body) ⇒ {
198                 self.gamma_push();
199                 self.gamma_insert(
200                     name.clone(),
201                     Channel(Box::new(Session(Box::new(s.clone())))),
202                 );
203                 let _ = self.session(body)?;
204                 self.gamma_pop();
205                 Ok(Unit)
206             }
207
208             NewChannel(name, None, _) ⇒ Err(Box::new(MissingAnnotationForChannel(name.clone()))),
209
210             Send(channel, value, cont) ⇒ {
211                 let chan_ty = self.cml(channel)?;
212                 let val_ty = self.cml(value)?;
213                 // Stable matching: no `box` pattern
214                 if let Channel(inner) = &chan_ty
215                     && let Session(sa) = inner.as_ref()
```

```
                    && let SessionAction::SendAction(expected, _cont_s) = &**sa
                {
                    if &val_ty ≠ expected {
                        return Err(Box::new(TypeMismatch(val_ty, expected.clone())));
                    }
                    // (Optional) if channel is a plain variable, you could
                    // shadow it in a new scope with the continuation S.
                    // We keep it simple here:
                    let _ = self.session(cont)?;
                    return Ok(Unit);
                }
                Err(Box::new(InvalidSendOperation))
            }

            Receive(channel, var, cont) ⇒ {
                let channel_type = self.cml(channel)?;
                if let Channel(inner) = &channel_type
                    && let Session(sa) = inner.as_ref()
                    && let SessionAction::ReceiveAction(expected, _cont_s) = &**sa
                {
                    self.gamma_push();
                    self.gamma_insert(var.clone(), expected.clone());
                    let _ = self.session(cont)?;
                    self.gamma_pop();
                    return Ok(Unit);
                }

                Err(Box::new(ExpectedSessionChannel))
            }

            Select(channel, label, cont) ⇒ {
                let channel_type = self.cml(channel)?;
                let lbl = match label {
                    CmlTerm::Val(Value::Label(s)) ⇒ s.clone(),
                    _ ⇒ return Err(Box::new(ExpectedSessionChoice)),
                };
                if let Channel(inner) = &channel_type
                    && let Session(sa) = inner.as_ref()
                    && let SessionAction::SelectAction(arms) = &**sa
                {
                    if let Some((_, _)) = arms.iter().find(|(l, _)| l == &lbl) {
                        let _ = self.session(cont)?;
                        return Ok(Unit);
                    } else {
                        return Err(Box::new(InvalidBranchSelection(
                            SessionAction::SelectAction(arms.clone()),
                            SessionAction::Var(lbl),
                        )));
                    }
                }

                Err(Box::new(ExpectedSessionChannel))
            }

            Offer(channel, branches) ⇒ {
                let channel_type = self.cml(channel)?;
                let expected: Vec<(String, SessionAction)> = if let Channel(inner) = &channel_type {
                    if let Session(sa) = inner.as_ref() {
                        if let SessionAction::BranchAction(arms) = &**sa {
                            arms.clone()
                        } else {
                            return Err(Box::new(ExpectedBranch));
                        }
                    } else {
                        return Err(Box::new(ExpectedSessionChannel));
                    }
                } else {
                    return Err(Box::new(ExpectedSessionChannel));
```

```
284                 };
285                 if branches.is_empty() {
286                     return Err(Box::new(EmptyBranchSet));
287                 }
288                 if branches.len() ≠ expected.len() {
289                     return Err(Box::new(BranchCountMismatch(
290                         branches.len(),
291                         expected.len(),
292                     )));
293                 }
294
295                 let exp_map: HashMap<_, _> = expected.into_iter().collect();
296                 let mut ret_tys = Vec::with_capacity(branches.len());
297                 for (lbl_term, body) in branches {
298                     let lbl = match lbl_term {
299                         CmlTerm::Val(Value::Label(s)) ⇒ s.clone(),
300                         _ ⇒ return Err(Box::new(ExpectedSessionChoice)),
301                     };
302
303                     let _next_s = exp_map.get(&lbl).ok_or_else(|| {
304                         Box::new(ValueNotInBranch(lbl_term.clone())) as Box<dyn Error>
305                     })?;
306
307                     let ty = self.session(body)?;
308                     ret_tys.push(ty);
309                 }
310
311                 let t0 = ret_tys[0].clone();
312                 if ret_tys.iter().any(|t| *t ≠ t0) {
313                     return Err(Box::new(BranchTypeMismatch(t0, ret_tys[1].clone())));
314                 }
315                 Ok(t0)
316             }
317
318             Parallel(p1, p2) ⇒ {
319                 let _ = self.session(p1)?;
320                 let _ = self.session(p2)?;
321                 Ok(Unit)
322             }
323
324             Replicate(p) ⇒ {
325                 let _ = self.session(p)?;
326                 Ok(Unit)
327             }
328
329             SVar(name) ⇒ self
330                 .sigma_lookup(name)
331                 .ok_or_else(|| Box::new(UndefinedSessionVar(name.clone())) as _),
332
333             SessionTerm::Rec(var, body) ⇒ {
334                 self.sigma_insert(var);
335                 let _ = self.session(body)?;
336                 Ok(Unit)
337             }
338
339             Link(c1, c2, cont) ⇒ {
340                 let t1 = self.cml(c1)?;
341                 let t2 = self.cml(c2)?;
342                 let (s1, s2) = match (&t1, &t2) {
343                     (Channel(a1), Channel(a2)) ⇒ match (a1.as_ref(), a2.as_ref()) {
344                         (Session(s1), Session(s2)) ⇒ (s1.clone(), s2.clone()),
345                         _ ⇒ return Err(Box::new(ExpectedChannelType)),
346                     },
347                     _ ⇒ return Err(Box::new(ExpectedChannelType)),
348                 };
349                 if !s1.is_dual_of(&s2) {
350                     return Err(Box::new(ChannelTypeMismatch(t1, t2)));
351                 }
```

```
                let _ = self.session(cont)?;
                Ok(Unit)
            }

            End ⇒ Ok(Unit),
        }
    }
}

#[derive(Debug, Clone, Default)]
pub struct Runtime {
    pub arena: SharedArena,
}

impl Runtime {
    pub fn new() -> Self {
        Runtime {
            arena: Arc::new(Mutex::new(Arena::new())),
        }
    }

    pub fn run(&mut self, term: SessionTerm) -> Result<(), Box<dyn Error + Sync + Send>> {
        let mut process = Process::new(term);
        let _ = eval_game(&mut process, &self.arena)?;
        Ok(())
    }

    pub fn arena_dot(&self) -> String {
        let ar = self.arena.lock().unwrap();
        ar.to_dot()
    }
}
```

1.6 Error Handling

```
use crate::channel::SessionAction;
use crate::syntax::{CmlTerm, Type};
use MemoryError::*;
use SessionError::*;
use TypeError::*;
use std::error::Error;
use std::fmt::{Display, Formatter};

#[derive(Debug)]
pub enum TypeError {
    UndefinedVariable(String),
    TypeMismatch(Type, Type),
    BranchTypeMismatch(Type, Type),
    MissingTypeAnnotation,
    MissingAnnotationForChannel(String),
    ConcurrentAccess(String),
    UndefinedType(String),
    UnsportedValueType,
    ExpectedChannel,
    ExpectedChannelType,
    ChannelTypeMismatch(Type, Type),
    ExpectedSessionChannel,
    ExpectedSessionChoice,
    ExpectedInteger,
    ExpectedFunction,
    ExpectedRecursiveLambda,
    ExpectedExponential,
    FunctionTypeMismatch(Type, Type),
    ExpectedFunctionType,
    ExpectedSessionType,
    ExpectedTensorType,
```

```
32      ExpectedBoolean,
33      ExpectedReference,
34      ExpectedReferenceType,
35      ExpectedExponentialType,
36      ExpectedBranch,
37      ExpectedPair,
38      InvalidParallelTerm,
39      EmptyBranchSet,
40      ValueNotInBranch(CmlTerm),
41      AddressIsRuntimeOnly,
42      NonPure,
43  }
44
45  #[derive(Debug)]
46  pub enum MemoryError {
47      ChannelExists(String),
48      ChannelNotDefined(String),
49      EmptyChannel(String),
50      AddressNotFound(usize),
51  }
52
53  #[derive(Debug)]
54  pub enum SessionError {
55      ProtocolViolation(SessionAction, SessionAction),
56      InvalidTransition(SessionAction, SessionAction),
57      InvalidBranchSelection(SessionAction, SessionAction),
58      BranchCountMismatch(usize, usize),
59      ExpectedSendOrReceive,
60      ExpectedSelect,
61      ExpectedOffer,
62      ProtocolDoesntAllowOffer,
63      ProtocolDoesntAllowSelect,
64      InvalidSendOperation,
65      InvalidReceiveOperation,
66      InvalidSessionChoice,
67      InvalidContinuation,
68      InvalidComposition(SessionAction, SessionAction),
69      CouldntFindPartner(String),
70      DualityMismatch(SessionAction, SessionAction),
71      AlreadyComposed,
72      StackMismatch(usize, usize),
73      InvalidRecursion(String),
74      IncompleteSession(String, SessionAction),
75      ChannelPending(String),
76      DisconnectedChannel(String),
77      InvalidBranchLabel(String),
78      UndefinedSessionVar(String),
79  }
80
81  #[derive(Debug)]
82  pub enum ArenaError {
83      InvalidJustification,
84      InvalidNode(usize),
85      InvalidPlayerSequence(usize),
86      OrphanNode(usize),
87      MissingCausalParent
88  }
89
90  #[derive(Debug)]
91  pub enum ParserError {
92      UnexpectedEof {
93          file: String,
94          line: usize,
95          column: usize
96      },
97      UnexpectedToken {
98          file: String,
99          expected: String,
```

```
100             found: String,
101             line: usize,
102             column: usize
103         },
104         TrailingInput {
105             file: String,
106             found: String,
107             line: usize,
108             column: usize
109         },
110         DuplicateLabel {
111             file: String,
112             label: String,
113             line: usize,
114             column: usize
115         },
116         EmptyBranchSet {
117             file: String,
118             line: usize,
119             column: usize
120         },
121         InvalidIntLiteral{
122             file: String,
123             text: String,
124             line: usize,
125             column: usize
126         },
127         ExpectedIdent {
128             file: String,
129             what: &'static str,
130             line: usize,
131             column: usize
132         },
133         ExpectedLabel {
134             file: String,
135             line: usize,
136             column: usize
137         },
138         ExpectedComma {
139             file: String,
140             line: usize,
141             column: usize
142         },
143         ExpectedCommType {
144             file: String,
145             line: usize,
146             column: usize
147         },
148         ExpectedEqual {
149             file: String,
150             line: usize,
151             column: usize
152         },
153         LexError{
154             file: String,
155             msg: String,
156             line: usize,
157             column: usize
158         },
159         IoError {
160             file: String,
161             msg: String
162         },
163     }
164
165     impl Error for TypeError {}
166
167     impl Error for MemoryError {}
```

```
168
169 impl Error for SessionError {}
170
171 impl Error for ArenaError {}
172
173 impl Error for ParserError {}
174
175 impl Display for TypeError {
176     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
177         match self {
178             UndefinedVariable(var) ⇒ write!(f, "Undefined variable: {}", var),
179             TypeMismatch(expected, found) ⇒ write!(
180                 f,
181                 "Type mismatch: expected {:?}, found {:?}",
182                 expected, found
183             ),
184             ChannelTypeMismatch(expected, found) ⇒ write!(
185                 f,
186                 "Channel type mismatch: expected {:?}, found {:?}",
187                 expected, found
188             ),
189             ConcurrentAccess(var) ⇒ write!(f, "Concurrent access to variable: {}", var),
190             FunctionTypeMismatch(expected, found) ⇒ write!(
191                 f,
192                 "Function type mismatch: expected {:?}, found {:?}",
193                 expected, found
194             ),
195             BranchTypeMismatch(expected, found) ⇒ write!(
196                 f,
197                 "Branches must have the same type: expected {:?}, found {:?}",
198                 expected, found
199             ),
200             ExpectedChannel ⇒ write!(f, "Expected a channel"),
201             ExpectedChannelType ⇒ write!(f, "Expected a channel type"),
202             ExpectedSessionChannel ⇒ write!(f, "Expected a session channel"),
203             ExpectedFunction ⇒ write!(f, "Expected a function"),
204             ExpectedFunctionType ⇒ write!(f, "Expected a function type"),
205             ExpectedSessionType ⇒ write!(f, "Expected a session type"),
206             ExpectedSessionChoice ⇒ write!(f, "Expected a session choice"),
207             ExpectedTensorType ⇒ write!(f, "Expected a tensor type"),
208             UnsportedValueType ⇒ write!(f, "Unsupported value type"),
209             ExpectedBoolean ⇒ write!(f, "Expected a boolean"),
210             ExpectedInteger ⇒ write!(f, "Expected an integer"),
211             ExpectedReference ⇒ write!(f, "Expected a reference"),
212             ExpectedReferenceType ⇒ write!(f, "Expected a reference type"),
213             ExpectedExponential ⇒ write!(f, "Expected an exponential"),
214             ExpectedExponentialType ⇒ write!(f, "Expected an exponential type"),
215             ExpectedBranch ⇒ write!(f, "Expected a branch"),
216             ExpectedPair ⇒ write!(f, "Expected a pair"),
217             EmptyBranchSet ⇒ write!(f, "Empty branch list"),
218             ValueNotInBranch(value) ⇒ write!(f, "Value {} not in branch", value),
219             UndefinedType(name) ⇒ write!(f, "Undefined type: {}", name),
220             NonPure ⇒ write!(f, "Trying to evaluate a non-pure term as pure"),
221             InvalidParallelTerm ⇒ write!(f, "Invalid parallel term"),
222             MissingTypeAnnotation ⇒ write!(f, "Missing type annotation"),
223             MissingAnnotationForChannel(name) ⇒ write!(f, "Missing type annotation for channel: {}", name
224             AddressIsRuntimeOnly ⇒ write!(f, "Address values are only available at runtime"),
225             ExpectedRecursiveLambda ⇒ write!(f, "Expected a recursive lambda"),
226         }
227     }
228 }
229
230 impl Display for MemoryError {
231     fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
232         match self {
233             ChannelExists(name) ⇒ write!(f, "Channel '{}' already exists", name),
234             ChannelNotDefined(name) ⇒ write!(f, "Channel '{}' not defined", name),
235             EmptyChannel(name) ⇒ write!(f, "Channel {} is empty", name),
```

```
236             AddressNotFound(loc) => write!(f, "Address {} not found", loc),
237         }
238     }
239 }
240
241 impl Display for SessionError {
242     fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
243         match self {
244             ProtocolViolation(expected, found) => write!(
245                 f,
246                 "Session protocol violation: expected {:?}, found {:?}",
247                 expected, found
248             ),
249             InvalidTransition(s1, s2) => {
250                 write!(f, "Invalid session transition: {:?} -> {:?}", s1, s2)
251             }
252             InvalidBranchSelection(expected, found) => write!(
253                 f,
254                 "Invalid branch selection: expected {:?}, found {:?}",
255                 expected, found
256             ),
257             BranchCountMismatch(expected, found) => write!(
258                 f,
259                 "Branch count mismatch: expected {}, found {}",
260                 expected, found
261             ),
262             ExpectedSendOrReceive => write!(f, "Expected send or receive action"),
263             ProtocolDoesntAllowOffer => write!(f, "Protocol doesn't allow offer"),
264             ProtocolDoesntAllowSelect => write!(f, "Protocol doesn't allow select"),
265             InvalidSessionChoice => write!(f, "Invalid session choice"),
266             InvalidSendOperation => write!(f, "Invalid send operation"),
267             InvalidReceiveOperation => write!(f, "Invalid receive operation"),
268             InvalidContinuation => write!(f, "Invalid continuation"),
269             DualityMismatch(p1, p2) => write!(
270                 f,
271                 "Protocol duality mismatch: {:?} is not dual of {:?}",
272                 p1, p2
273             ),
274             InvalidComposition(p1, p2) => write!(
275                 f,
276                 "Invalid session composition: {:?} cannot be composed with {:?}",
277                 p1, p2
278             ),
279             AlreadyComposed => write!(f, "Session already composed"),
280             StackMismatch(s1, s2) => write!(f, "Session protocol stack mismatch: {} ≠ {}", s1, s2),
281             InvalidRecursion(name) => write!(f, "Invalid recursion: {}", name),
282             CouldntFindPartner(name) => write!(f, "Couldn't find partner for {}", name),
283             IncompleteSession(name, current) => write!(f, "Incomplete session {}: {:?}", name, current),
284             ChannelPending(name) => write!(f, "Channel {} is pending", name),
285             DisconnectedChannel(name) => write!(f, "Channel {} is disconnected", name),
286             InvalidBranchLabel(label) => write!(f, "Invalid branch label: {}", label),
287             ExpectedOffer => write!(f, "Expected an offer action"),
288             ExpectedSelect => write!(f, "Expected a select action"),
289             UndefinedSessionVar(name) => write!(f, "Undefined session variable: {}", name),
290         }
291     }
292 }
293
294
295 impl Display for ArenaError {
296     fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
297         use ArenaError::*;
298         match self {
299             InvalidJustification => write!(f, "Invalid justification"),
300             InvalidNode(id) => write!(f, "Invalid node: {}", id),
301             InvalidPlayerSequence(id) => write!(f, "Invalid player sequence at node {}", id),
302             OrphanNode(id) => write!(f, "Orphan node: {}", id),
303             MissingCausalParent => write!(f, "Missing causal parent"),
```

```
        }
    }
}

impl Display for ParserError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        use ParserError::*;
        match self {
            UnexpectedEof { file, line, column } =>
                write!(f, "{file}:{line}:{column}: unexpected end of input"),
            UnexpectedToken { expected, found, file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected {expected}, found {found}"),
            TrailingInput { found, file, line, column } =>
                write!(f, "{file}:{line}:{column}: trailing input starting at {found}"),
            DuplicateLabel { label, file, line, column } =>
                write!(f, "{file}:{line}:{column}: duplicate label `{label}`"),
            EmptyBranchSet { file, line, column } =>
                write!(f, "{file}:{line}:{column}: empty branch set"),
            InvalidIntLiteral { text, file, line, column } =>
                write!(f, "{file}:{line}:{column}: invalid integer literal `{text}`"),
            ExpectedIdent { what, file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected {what}"),
            ExpectedLabel { file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected a label"),
            ExpectedComma { file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected a ',' between channel and argument"),
            ExpectedCommType { file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected a <Type> after send/receive"),
            ExpectedEqual { file, line, column } =>
                write!(f, "{file}:{line}:{column}: expected '='"),
            LexError { msg, file, line, column } =>
                write!(f, "{file}:{line}:{column}: {msg}"),
            IoError { file, msg } =>
                write!(f, "{file}: {msg}"),
        }
    }
}

#[derive(Debug)]
pub struct Anyhow(pub &'static str);

impl Error for Anyhow {}
impl Display for Anyhow {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.0)
    }
}

```