DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

GONÇALO MARTINS LOURENÇO

BSc in Computer Science and Engineering

EXTENDING A SESSION-BASED CONCURRENT LANGUAGE BEYOND LINEARITY

MASTER IN MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

NOVA University Lisbon
September, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

EXTENDING A SESSION-BASED CONCURRENT LANGUAGE BEYOND LINEARITY

GONÇALO MARTINS LOURENÇO

BSc in Computer Science and Engineering

Adviser: Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, NOVA University Lisbon

MASTER IN MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

NOVA University Lisbon
September, 2025

EXTENDING A SESSION-BASED CONCURRENT LANGUAGE BEYOND LINEARITY

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.18) [1].

For all those who believed in me,
and to those who doubted.

Acknowledgements

I would like to start by thanking my advisor, Bernardo Toninho, for his patience throughout the entire process, particularly during the numerous doubts that arose, and above all, for the opportunity to work with someone so knowledgeable in topics I knew so little about.

I would also like to thank my family, who have always supported me and encouraged me to keep going, especially my parents, Jorge and Lídia, for believing in me and helping me to follow my dreams.

I must also mention my girlfriend, Joana, who accompanied me throughout my journey and provided invaluable support along the way. She was the one who put up with my every step and showed me that it was possible to keep moving forward.

Finally, I thank my friends and colleagues who walked a path similar to mine and could share my burdens.

I thank you with all my heart!

Abstract

As technology has evolved, applications have become increasingly complex. They have evolved not only in the number of operations they perform, but also in the number of processes by which these operations are distributed. This phenomenon can be observed in all types of applications, regardless of their functionality and consumer. As such, the importance of concurrency, management mechanisms, and communication between competing processes has increased.

Due to its importance, multiple approaches have emerged. *Session types*, being one of those, introduced the notion of channels through which different running processes communicate. This protocol also forces these channels to follow a predetermined protocol, allowing the processes interacting with the channels to know exactly what type of value they must send or receive.

Logical session types are even more strict than *session types* but come with the benefit solving problems like *starvation* and *deadlocks*.

In prior work, *SILL* was created as a language that employs strict concurrency control mechanisms, based on logical session types, to ensure session fidelity. This work builds upon the compiler of *SILL*. It aims to relax some of these strict restrictions, while conscientiously considering the dangers that come with such relaxation, in order to make the language more expressive.

Expanding *SILL* is achieved by implementing the concept of *shared channels*, which can be shared and accessed across multiple processes, introducing some risks but allowing for greater freedom when writing programs.

Finally, after testing the language improvements, it can be concluded that despite the risks and added attention when writing programs, performance was not affected when using shared processes instead of the previously supported linear ones.

Keywords: Concurrency, Session Types, Logical Session Types, SILL, GO, Manifest Sharing

Resumo

Com a evolução da tecnologia, aplicações têm se tornado cada vez mais complexas. Elas evoluíram não apenas no número de operações que realizam, mas também no número de processos pelos quais essas operações são distribuídas. Este fenómeno pode ser observado em todos os tipos de aplicações, independentemente da sua funcionalidade ou consumidor. Assim, a importância da concorrência, dos mecanismos de gestão e da comunicação entre processos concorrentes registou um grande crescimento.

Devido à sua importância, surgiram várias abordagens. Os *tipos de sessão*, sendo uma delas, introduziram a noção de canais através dos quais diferentes processos em execução comunicam. Este protocolo também obriga esses canais a seguir um protocolo predeterminado, permitindo que os processos que interagem com os canais saibam exatamente que tipo de valor devem enviar ou receber.

Os *tipos de sessão lógica* são ainda mais rigorosos do que os *tipos de sessão*, mas têm a vantagem de resolver problemas como *starvation* e *deadlocks*.

Em trabalhos anteriores, o *SILL* foi criado como uma linguagem que emprega mecanismos rigorosos de controlo de concorrência, baseados em tipos de sessão lógica, para garantir a fidelidade da sessão. Este trabalho baseia-se no compilador do *SILL*. O objetivo é flexibilizar algumas dessas restrições rígidas, considerando conscientemente os perigos que tal flexibilização acarreta, a fim de tornar a linguagem mais expressiva.

A expansão do *SILL* é alcançada através da implementação do conceito de *canais partilhados*, que podem ser partilhados e acedidos por vários processos, introduzindo alguns riscos, mas permitindo maior liberdade na escrita de programas.

Finalmente, após testar as melhorias da linguagem, pode-se concluir que, apesar dos riscos e da atenção adicional necessária ao escrever programas, o desempenho não foi afetado ao usar processos partilhados em vez dos lineares anteriormente suportados.

Palavras-chave: Concurrência, Tipos de Sessão, Tipos de sessão Lógicos, SILL, GO, Manifest Sharing

Contents

1 Introduction

As the complexity of programs continues to rise, so does the importance of sharing the workload across multiple processes. Running concurrently while working on different tasks, these processes need to communicate safely with one another.

There are several challenges when writing concurrent programs. While the language addresses some aspects, it is challenging to share resources across multiple processes safely and securely. Maintenance and scaling, or editing, these programs is also challenging, given the varying results that multiple executions can yield with no clear way to debug them. In addition to these difficulties, it is typically the programmers' responsibility to ensure the program is free from deadlocks, starvation, and other similar issues.

Shared memory and *message passing* are some of the first paradigms adopted in order to provide communication in concurrent programs. *Shared memory* is, as the name implies, a memory address known to multiple processes trying to interact with each other that they can write to and read. Being one of the fastest methods of communication due to its simplicity, it comes with several restrictions. Since this method does not guarantee it, the developers must ensure that the operations are safe enough not to compromise the program itself (concurrent reading and writing can lead to inconsistent values either read or written). Another issue that arises is the lack of privacy and control over the data that a process shares. There is no guarantee that it reaches the correct process without interference from others.

Message passing, despite not being perfect, presents solutions to the previous issues by introducing the concept of channels. This level of abstraction helps developers by automating low-level concurrency control, eliminating the need to worry about low-level mechanisms such as locks or semaphores.

To reduce communication errors, the channels introduced in *message passing* can be restricted by following a specific protocol. *Session types* will make sure that the type of the object or value sent is the same as the type expected to be received.

Linear session types are then introduced by refining *message passing* even further in

order to prevent issues like *deadlocks* or *starvation*, however, it comes with the price of expressivity.

Finally, *SILL*, which is the starting point of this work ([2]), is a practical implementation of *linear session types*. To meet all the requirements, the language becomes quite restrictive, however. This work is about conscientiously expanding *SILL* in order for it to become more expressive.

Our approach aligns with previous work on *Manifest Sharing* ([3]), which integrates shared state into session-typed communication while preserving session fidelity. Other related approaches explore extensions with mutable state ([4]), differential linear logic ([5]), or propositions-as-types for shared state ([6]), demonstrating different ways of combining linear reasoning with mutable resources.

1.1 Document Outline

This document continues in chapter 2, presenting the background related to this work. It starts by reviewing *session types* and *logical session types*. Then, it moves to *SILL*, the starting point of this project, before finally reviewing *manifest sharing*, which introduced the ideas implemented in this work.

chapter 3 examines the syntax of the custom language and expands it to include shared types. The chapter continues to explain the workings of the typechecker and the compiler, providing examples of linear and shared programs.

Finally, chapter 4 evaluates the performance of this work before the brief conclusion in chapter 5.

1.2 Repository

The code of this project is in a public GitHub repository [7]. The project has a folder named `sessint` where the compiler itself is and all the examples used in the figures are in the `results` folder. Inside this folder for each of the example there is the original `.prog` file with the custom program as well as `.go` file with the result.

2 Background and Related Work

For modern computing, concurrency is an absolute necessity, and it is evident in tasks ranging from the simplest to the most demanding. It involves organizing computational work into independent segments, with programs composed of processes that execute independently of one another. Following Andrew Gerrand's words, it is up to programmers to manage these processes; however, programming languages also play a crucial role in determining the mechanisms they support and the implications of these choices.

As a consequence, it is common for processes to compete for resources. This fact, coupled with the need for data exchange between different processes, led to the development of inter-process communication mechanisms.

This communication, this competition for resources, was initially achieved mainly through *shared memory*. It was the most natural solution and remains in use to this day.

2.0.1 Shared Memory

When we discuss *shared memory*, we are referring to the process of sharing data by writing it into a specific region of an address space, which other processes can then access and process as needed. The primary advantages of this approach are its speed and simplicity. With shared memory, a process allocates space in memory and records its address. The process then writes to this address, enabling other authorized processes to read from and write to it.

Despite its conceptual soundness and more straightforward implementation, *shared memory* concurrency has several drawbacks. With this concurrency mechanism, programmers must ensure that processes do not access shared addresses simultaneously. While concurrent reads are generally unproblematic, simultaneous reads and writes can corrupt memory or result in incorrect data. To prevent these issues, developers must use memory locking mechanisms to block other processes from interfering with the data address they intend to use. Each time a process needs to write data, it must lock the address, causing other processes to wait for access to that memory. Suppose a process can lock some

memory while others are waiting to access it. In that case, shared memory can lead to programs with deadlocks, where a group of processes mutually wait on each other and cannot proceed with execution.

Another limitation of this mechanism is the lack of control over data ownership. When sharing data, we write it without controlling who reads or modifies it. This disadvantage is contentious, as it is also *shared memory*'s most significant advantage. The absence of data management concerns contributes to its simplicity and potential for improved speeds.

Understanding the challenges and limitations of *shared memory*, there was a need to develop an alternative communication method that would facilitate straightforward synchronization between various processes. This automatic synchronization, however, incurs a runtime cost that is a consequence of automating lock mechanisms.

2.0.2 Message Passing

With the need that arose for a more sophisticated communication mechanism, *message passing*, wherein data, in the form of messages, are exchanged from one process to another. In the present day, this mechanism, which offers a higher-level abstraction for communication, is recognized for having two significant advantages, despite its added complexity and potentially higher time cost compared to shared memory.

The first advantage of *message passing* is that it provides a higher level of abstraction, helping developers avoid mistakes originating from low-level implementations and their associated burdens. The synchronization is implemented by the communication mechanism, freeing developers to abstract the concurrency safety aspects. As a consequence, each process can send or exchange messages without worrying about locking and mutual exclusion of the communication medium, or about corrupting data or receiving corrupted data, since each message is treated separately and delivered directly to its intended receiver.

A frequently utilized abstraction involves conceptualizing *message passing* in conjunction with *channels*. To establish a communication link between two processes, a channel is created between the sender and the receiver, ensuring that no other process can interfere with the communication. This approach simplifies the understanding of a message being transmitted securely and without interference.

Although *message passing* offers solutions to specific issues associated with *shared memory*, some problems persist. *Deadlocks* remain a concern, as they can still occur in message passing. Nevertheless, *message passing* provides several advantages in this domain, primarily because communication is inherently end-to-end.

Previously, it was the developers' responsibility to handle errors arising from data type mismatches between sent and received messages. However, mechanisms now exist that enable a language to be type-safe. By verifying the types of messages transmitted through channels, it is possible to implement "type-safe message passing".

2.1 Session Types

In broad terms, *session types* [8, 9] integrate *message passing* with mechanisms to maintain session fidelity. The objective of *session types* is to offer a clear and systematic method for defining complex interaction behaviors at a high level of abstraction. Additionally, they provide a formal foundation for verification and, through their type system, ensure the correctness of the exchanged messages.

In order to achieve this objective, *session types* introduce the concept of *sessions*, which represent a sequence of binary operations that form a program. Communication within sessions occurs through a private port known as a *channel*.

To facilitate effective communication within these sessions, three fundamental communication primitives are employed: *delegation*, *labelled choice*, and *value passing*.

Value passing allows session participants to exchange fundamental data values, such as integers and strings. *Labelled choice* enables the offering of multiple options within a communication protocol, along with the corresponding methods for responding to these options. *Delegation* allows a programmer to dynamically distribute a single session among multiple processes in an organized manner by sending a channel to another process.

Ultimately, the type-checker ensures that the sequence of interactions within a channel adheres to a specific type or protocol. A fundamental aspect of this structuring technique is the establishment of a basic type discipline for communication primitives. For example, from the caller's perspective, a procedural call follows an output-input communication pattern, while the callee should follow an input-output communication pattern. This type of discipline is of considerable practical importance, as mismatched interaction patterns across processes are a primary source of errors in communication-based programming.

Due to its usefulness, *session types* have been integrated into various programming languages [10–14] that express such protocols [3].

When adding *session types* to a linear logic foundation, it is possible to create a language [4, 15–17] like the one we are delving more deeply into subsection 2.1.1 [3].

2.1.1 Our starting point: SILL

This work is built upon the research conducted by Toninho *et al.* [18], which introduces a session-typed language known as SILL. SILL is a higher-order functional language that incorporates session types into the functional programming paradigm. It aims to provide a high-level abstraction for defining communication protocols and ensuring their correctness through a robust type system.

The syntax of SILL is derived from session types, enabling the specification of communication protocols as types. This syntax facilitates the definition of concurrent processes that interact via channels. The language encompasses constructs for input and output operations, branching, and recursion. The type system guarantees that inter-process communication adheres to the specified protocol, thereby preventing errors such as deadlocks

and type mismatches.

Before we dive deeper into this language, however, we must first take a look at the syntax of session types and their corresponding usages in a concurrent language. As such, Figure 2.1 demonstrates the type system and Figure 2.2 process syntax according to SILL. They follow Toninho *et al.* [18] and allow us to specify small communication protocols as types.

τ	::=	$nat \mid bool \mid ...$	(ordinary functional types)
A, B, C	::=	$\tau \supset A$	input value of type τ and continue as A
	$\mid$	$\tau \wedge A$	output value of type τ and continue as A
	$\mid$	$A \multimap B$	input channel of type A and continue as B
	$\mid$	$A \otimes B$	output fresh channel of type A and continue as B
	$\mid$	$\&\{\overline{l_j : A_j}\}$	offer choice between l_j and continue as A_j
	$\mid$	$\oplus\{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
	$\mid$	$\mathbf{1}$	terminate
	$\mid$	$\mu X.A \mid X$	recursive process type

Figure 2.1: Type Syntax

In Figure 2.1, τ ::= $nat \mid bool \mid ...$ represents the functional types; it consists of fundamental data types such as booleans and natural numbers, in addition to function types. Standard constructs, including products and sums, are excluded [18].

Session types are denoted by A and B. The types $\tau \supset A$ and $\tau \wedge A$ denote value input and output, respectively, within the context of session types. More specifically, $\tau \supset A$ symbolizes the type of a channel that receives a value of type τ, which will subsequently behave as type A. Conversely, $\tau \wedge A$ represents the type of a channel that sends a value of type τ, which will behave as type A. These examples illustrate the duality and complementarity inherent in the two types. These properties arise from the use of sessions, where each end of a session has a specific role that is only feasible with its counterpart. For instance, one end is responsible for sending a message, while the other end is responsible for receiving it.

$A \multimap B$ denotes the type of a channel that receives a channel of type A and subsequently behaves as type B. Conversely, $A \otimes B$, its complementary type, represents a channel that sends a channel of type A and behaves as type B.

The types $\&\overline{l_j : A_j}$ and $\oplus\overline{l_j : A_j}$ are used to offer a choice between labels l_j, with the complementary type being used to select between these labels correspondingly before defining the type A_j for the channel.

Lastly, $\mathbf{1}$ and $\mu X.A \mid X$ do not exhibit the previously discussed properties, as will be shown. The type $\mathbf{1}$ allows both processes in the session to terminate it in the same manner.

The session type $\mu X.A$ is a recursive session type, where the type variable X represents the recursive type itself within A.

As previously mentioned, a session is the fundamental concept in a *session types* system. A session serves as an abstraction for defining interactions and consists of a sequence of reciprocal exchanges between two participants, potentially involving branching and recursion. A channel refers to the port through which session communications are conducted. Each time a session is initiated, a new channel is created for use in these communications [8]. However, we have not yet explored how to create, define, or utilize such sessions. For these purposes, we refer to the syntax in Figure 2.2.

P, Q	$::=$	$c \leftarrow Q; P_c$	compose process computed by Q in parallel with P_c, communicating along fresh channel c
	$\mid$	$x \leftarrow \texttt{input}\ c; Q_x$	input a value x along channel c
	$\mid$	$\_ \leftarrow \texttt{output}\ c\ v; P$	output value of Q along channel c
	$\mid$	$d \leftarrow \texttt{input}\ c; Q_d$	input channel d along channel c
	$\mid$	$\_ \leftarrow \texttt{output}\ c\ d; Q$	output a fresh channel d along c
	$\mid$	$\texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j}$	branch on selection of l_j along c
	$\mid$	$\_ \leftarrow c.l_j; P$	select label l_j along c
	$\mid$	$\_ \leftarrow \texttt{wait}\ c; P$	wait for closure of c
	$\mid$	$\texttt{close}\ c$	close channel c and terminate
	$\mid$	$def\ D\ in\ P$	recursion
D	$::=$	$X_1 = P_1\ and\ ...\ X_n = P_n$	Declaration for Recursion

Figure 2.2: Syntax for Process Expressions

In Figure 2.2, we present the syntax required to define concurrent processes. Specifically, the expression $c \leftarrow Q; P_c$ enables us to compose processes Q and P in parallel, facilitating communication through a newly created channel c.

Many of the subsequent expressions have corresponding complementary operations, similar to those in Type Syntax, Figure 2.1.

Furthermore, the types of channels created in each process expression can be readily aligned with the type syntax presented in Figure 2.1. In the subsequent chapter, we will conduct an in-depth analysis of this alignment and examine its implications within the context of the language we are developing, supported by real-world examples.

Referring back to Figure 2.2, the expression $x \leftarrow \texttt{input}; c; Q_x$ signifies the reception of a value x from process Q_x via channel c. Conversely, the expression $\_ \leftarrow \texttt{output}; c; v; P$ represents the complementary operation, enabling the transmission of a value v through channel c.

The operation $d \leftarrow \texttt{input}; c; Q_d$ functions similarly to the previous input operation, with the distinction lying in the type of data transferred. Instead of receiving a value, this operation receives a channel. Its complementary operation, $\_ \leftarrow \texttt{output}; c; d; Q$, enables a process to send a newly created channel d.

The subsequent operations, $\texttt{case}; c; \texttt{of}; \overline{l_j \Rightarrow P_j}$ and $\_ \leftarrow c.l_j; P$, illustrate the mechanisms for branching a selection and choosing from the branched labels, l_j.

The operations $\leftarrow\texttt{wait}; c; P$ and $\texttt{close}; c$ signify waiting for a channel to close and closing a channel, respectively.

Finally, $def; D; in; P$ facilitates the definition of (potentially mutually recursive) processes.

2.1.2 Examples

To demonstrate the syntax and types detailed above in Example 2.1 and Example 2.2 below. We begin with a simple recursive example and then proceed to a more complex scenario, where a client utilizes a more practical version of the function from Example 2.1.

Example 2.1 (Simple Incremental Function)**.**

$$nats_1(x) \;=\; output\ c\ x;\ nats_1(x+1)$$

$$c : \mu X.\ nat\ /\backslash\ X$$

This example begins with the creation of a simple, incremental, recursive function named *nats*. This function takes an argument x of type *nat* and performs two actions. First, *nats* sends x through channel c, and then it recursively calls itself with its argument incremented by one, as illustrated in $nats(x+1)$.

Having examined our function in detail, it is pertinent to define c. As mentioned, c is a channel used to send x, which is an argument of the *nats* function of type *nat*. Therefore, our channel must also be of type *nat*.

△

Example 2.1 is intentionally simplified to facilitate a clearer understanding of our syntax. However, this simplicity comes at the expense of practical applicability. The initial version of the function *nats* essentially creates an infinitely running session. If we instantiate it as $c \leftarrow nats'(0);$, we obtain a channel c through which we must receive all natural numbers starting from 0, as there is no mechanism to signal the termination of the session.

Presented below is an alternative definition, $nats_2$, which provides the option to either emit the following natural number or terminate:

Example 2.2 (Client Bounded Incremental Function)**.**

$$
\begin{aligned}
nats_2(x) \;=\; case\ c\ of \quad & \overline{next \;\Rightarrow\; output\ c\ x;\ nats_2(x+1)} \\
& \overline{done \;\Rightarrow\; close\ c}
\end{aligned}
$$

$$
\begin{aligned}
c : \mu X. \quad & \&(next \;\Rightarrow\; nat \;/\backslash\; X \\
& \quad done \;\Rightarrow\; 1)
\end{aligned}
$$

$$
\begin{aligned}
client \;=\; \quad & c.l_{next};\ n_1 \;\leftarrow\; input\ c; \\
& c.l_{next};\ n_2 \;\leftarrow\; input\ c; \\
& c.l_{done};\ wait\ c;\ close\ c;
\end{aligned}
$$

$$
c \;\leftarrow\; nats_2(0);
$$

While our previous example described a process that executed indefinitely, the definition of $nats_2$ encodes a variation that continuously offers the option to terminate. It begins by analyzing the state of channel c. Suppose the channel corresponds to the label $c.l_{next}$. In that case, it proceeds by sending the value of parameter x through channel c and then recursively calls itself with the parameter value incremented by one ($nats(x+1)$).

In this example, the type of channel c offers a choice between the label $c.l_{next}$, where it behaves as a value of type nat, and $c.l_{done}$, where the value 1 of *terminate* is expected.

Our client is straightforward: it will utilize channel c twice while it remains branched with the label $c.l_{next}$, reading its value twice. Following this, it will switch the label of channel c to $c.l_{done}$ and wait for it to close ($wait\ c;$) on the other side. Finally, it will close channel c on its end as well ($close\ c;$).

△

Example 2.1 and Example 2.2 illustrate the ease of implementing a *session type* program using the presented syntax. At this juncture, it may be instructive to examine some of the limitations of *session types*.

Specifically, the session typing discipline permits the composition of processes that may lead to deadlocks, as demonstrated in Example 2.3.

Example 2.3 (Client deadlock)**.**

$$
\begin{aligned}
client_1 \;=\;& c_1n_1 \leftarrow input\ c;\\
& \_ \leftarrow output\ d\ 1;\\
& close\ c;\\
& wait\ d;\\
& close\ d;
\end{aligned}
$$

$$
\begin{aligned}
client_2 \;=\;& c_2n_1 \leftarrow input\ d;\\
& \_ \leftarrow output\ c\ 2;\\
& close\ d;\\
& wait\ c;\\
& close\ c;
\end{aligned}
$$

Where $client_1$ utilizes channels c and d in accordance with the session types $\mu X.\ nat \supset X$ and $\mu X.nat \wedge X$, respectively; and $client_2$ employs channels c and d in a dual manner (swapping inputs with outputs).

In this example, we are presented with two channels, c and d, which exhibit similar types and behaviors, allowing the bidirectional transfer of nat values. To utilize these channels, we have two clients, $client_1$ and $client_2$.

$client_1$ begins by attempting to read from channel c into the variable c_1n_1. Subsequently, it writes the value 1 to channel d, closes channel c from which it initially read, waits for channel d to be closed on the other end, and then closes its own connection to it.

Conversely, $client_2$ performs the same operations but with the channels reversed. When reading, $c_2n_1 \leftarrow input\ d$;, it reads from channel d. It then writes to channel c, $\_ \leftarrow output\ c\ 2$;, and finally, it proceeds to close its channels. It closes d, waits for channel c to be closed by $client_1$, and then closes its own end of channel c.

△

When a process P, within its current environment, attempts to execute an input or output operation for which there is no corresponding counterpart (output or input operation, respectively), this process P becomes stuck. P stagnates while awaiting communication, thereby preventing it from progressing to the next state. A *deadlock* occurs if all existing processes are in this situation, each waiting to communicate with another, and are consequently unable to proceed.

In Example 2.3, both $client_1$ and $client_2$ exhibit well-typed behavior. Each client reads a value from one channel, writes another value to a different channel, and subsequently closes its connections to these channels. The issue arises during execution when both processes run concurrently. $client_1$ and $client_2$ both begin by reading from their respective channels. However, while $client_1$ attempts to read a value from channel c, $client_2$ is simultaneously trying to read from channel d. Since neither process will write to its respective channel, they will both remain in a perpetual state of waiting.

With the framework discussed above, a wide range of applications can be developed. Compared to traditional *message passing* systems, systems implementing *session types* ensure that every action within a session (in a channel) adheres to session fidelity (within their respective protocols). The *type-checker* consistently verifies that types are correctly followed, making it a superior choice to its predecessor without excessively limiting the set of programs that can be created. However, since this mode of communication cannot guarantee that whenever we attempt to read from one channel, the other process has already written the intended message, the deadlock issue previously mentioned remains present in *session type* systems, as illustrated in Example 2.3.

2.2 Logical Session Types

While numerous type systems for channel-based concurrency have utilized various forms of linear channel usage [19], including the initial formulations of session types [8, 9], the relationship between these systems and linear logic was only uncovered through the work of Caires and Pfenning [15]. Their research demonstrates the feasibility of a propositions-as-types interpretation of linear logic, wherein propositions are construed as types, proofs are viewed as (typed) processes, and proof simplification is understood as inter-process communication.

The *logical session type* model introduced by Caires and Pfenning [15] is grounded in Dual Intuitionistic Linear Logic (DILL). It provides a direct interpretation of linear logic propositions as type processes, as illustrated below for both the *cut* and *conjunction* rules.

In intuitionistic linear logic, the cut rule is written as:

$$\frac{\Gamma, \Delta \vdash A \qquad \Gamma, \Delta', A \vdash C}{\Gamma, \Delta, \Delta' \vdash C}\ \textsc{cut}$$

$$\frac{\Gamma, \Delta \vdash P :: x : A \qquad \Gamma, \Delta', x : A \vdash Q :: z : C}{\Gamma, \Delta, \Delta' \vdash x \leftarrow \mathit{spawn}\ P; Q :: z : C}\ T\textsc{cut}$$

The *logical session type* model introduced by Caires and Pfenning [15] is grounded in Dual Intuitionistic Linear Logic (DILL). It provides a direct interpretation of linear logic propositions as type processes, as illustrated below for both the *cut* and *conjunction* rules.

The *cut* rule is regarded as one of the most fundamental logical principles. It encapsulates the concept of "reasoning from lemmas," asserting that a proof of C can be directly constructed by combining a proof of A (the "lemma") with another proof that assumes A to establish C [4].

From a practical perspective, the cut principle allows for the parallel composition of two processes, P and Q, utilizing a single (linear) channel. Process P provides the behavior A along channel x, while process Q leverages this behavior to deliver C. In this composition, the channel x remains local to both processes.

In linear logic, the Additive Conjunction rule signifies the alternative availability of resources and is expressed as follows:

$$\frac{\Gamma, \Delta \vdash A \quad \Gamma, \Delta \vdash B}{\Gamma, \Delta, \Delta' \vdash A \& B} \,\&\mathrm{R}$$

$$\frac{\Gamma, \Delta, x : A \vdash C}{\Gamma, \Delta, x : A \& B \vdash C} \,\&L_1 \qquad \frac{\Gamma, \Delta, x : B \vdash C}{\Gamma, \Delta, x : A \& B \vdash C} \,\&L_2$$

The process interpretation of the Additive Conjunction manifests as a form of binary branching. A process that implements sessions of types (A) and (B) offers a choice between these two sessions. Consequently, to utilize such a session, one must select the appropriate alternative:

$$\frac{\Gamma, \Delta \vdash P_1 :: z : A \quad \Gamma, \Delta \vdash P_2 :: z : B}{\Gamma, \Delta \vdash \mathit{case}\ z\ \mathit{of}\ [\overline{\mathsf{inl} \Rightarrow P_1},\ \overline{\mathsf{inr} \Rightarrow P_2}]} \,T\&\mathrm{R}$$

$$\frac{\Gamma, \Delta, x : A \vdash Q :: z : C}{\Gamma, \Delta, x : A \& B \vdash x.\mathsf{inl}; P :: z : C} \,T\&L_1$$

$$\frac{\Gamma, \Delta, x : B \vdash Q :: z : C}{\Gamma, \Delta, x : A \& B \vdash x.\mathsf{inr}; P :: z : C} \,T\&L_2$$

Similar to the interpretations of the cut rule and additive conjunction, there are corresponding interpretations for Linear Implication and various other constructs [4]. However, these interpretations fall outside the primary scope of this project.

The key idea is that by interpreting linear logic rules, we can transfer their beneficial logical properties to session types. By directly interpreting linear logic rules as types within a session type system, we can ensure that a process is free from deadlocks and, if finite, will eventually terminate.

Given that *logical session types* extend *session types*, we can reexamine Example 2.1, Example 2.2, and Example 2.3 from a logical perspective.

With *logical session types*, the first two examples (Example 2.1 and Example 2.2) would still function correctly. However, Example 2.3 would not result in a deadlock, as it would fail to compile. The compiler would detect that the clients are using multiple channels for communication, resulting in an error. During compilation, the compiler must construct a tree-like structure of the interactions between processes to ensure there are no cycles, as cycles could lead to deadlocks.

In essence, *logical session types* constrain the set of permissible programs by excluding any potential scenarios that could lead to a deadlock. While this restriction ensures that

logical session types are inherently deadlock-free, it also reduces the language's expressiveness, thereby excluding some cases that are deadlock-free.

2.3 SILL

A practical implementation of the preceding chapters is the *SILL* language [2].

In the preceding sections, the syntax employed was based on the *SILL* language. This choice was made not only for consistency, as it will underpin the proposed work, but also because it directly applies the topics discussed earlier. However, the syntax presented thus far is not exhaustive.

τ, σ	$::=$	$\tau \to \sigma \mid \ldots \mid \forall t.\, \tau \mid \mu t.\, \tau \mid t$	(ordinary functional types)
	$\mid$	$\{a{:}A \leftarrow \overline{a_i{:}A_i}\}$	process offering A along channel a, using channels a_i offering A_i
A, B, C	$::=$	$\tau \supset A$	input value of type τ and continue as A
	$\mid$	$\tau \wedge A$	output value of type τ and continue as A
	$\mid$	$A \multimap B$	input channel of type A and continue as B
	$\mid$	$A \otimes B$	output fresh channel of type A and continue as B
	$\mid$	$\&\{\overline{l_j : A_j}\}$	offer choice between l_j and continue as A_j
	$\mid$	$\oplus\{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
	$\mid$	$\mathbf{1}$	terminate
	$\mid$	$\mu X.A \mid X$	recursive process type
a	$::=$	c	linear channels
	$\mid$	$!u$	shared channels

Figure 2.3: SILL's Type Syntax

In contrast to the observations in Figure 2.1, Figure 2.3 demonstrates that, beyond the base types, products, and sums excluded for clarity, the functional types also encompass recursive and polymorphic types. The contextual monadic type $a{:}A \leftarrow \overline{a_i{:}A_i}$, which specifies the type of a process expression offering session A along channel a while utilizing channels a_i at types A_i, distinguishes this system. The process types largely remain consistent with previous definitions; however, Figure 2.3 introduces a new type of channel at the bottom. Previously, channels referred to as linear channels were denoted by c or d. Now, the type of shared channel is introduced, denoted by $!u$ or $!v$.

The most significant changes are concentrated in the syntax for monadic process expressions. In $M, N ::= \lambda x{:}\tau.\, M_x \mid M, N \mid \texttt{fix}, x.\, M_x \mid \ldots$, the terms M and N include their

M, N	$::=$	$\lambda x{:}\tau.\, M_x \mid M\, N \mid \mathtt{fix}\, x.\, M_x \mid \ldots$	(usual functional constructs)
	$\mid$	$a \leftarrow \{P_{a,\overline{a_i}}\} \leftarrow a_1, \ldots, a_n$	process providing a, using $a_1, \ldots, a_n$
P, Q	$::=$	$a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a$	compose process computed by M in parallel with P_a, communicating along fresh channel a
	$\mid$	$x \leftarrow \mathtt{input}\ c; Q_x$	input a value x along channel c
	$\mid$	$\_ \leftarrow \mathtt{output}\ c\ M; P$	output value of M along channel c
	$\mid$	$d \leftarrow \mathtt{input}\ c; Q_d$	input channel d along channel c
	$\mid$	$\_ \leftarrow \mathtt{output}\ c\ (d \leftarrow P_d); Q$	output a fresh channel d along c
	$\mid$	$\mathtt{case}\ c\ \mathtt{of}\ \overline{l_j \Rightarrow P_j}$	branch on selection of l_j along c
	$\mid$	$\_ \leftarrow c.l_j; P$	select label l_j along c
	$\mid$	$\_ \leftarrow \mathtt{wait}\ c; P$	wait for closure of c
	$\mid$	$\mathtt{close}\ c$	close channel c and terminate
	$\mid$	$\mathtt{fwd}\ c_1\ c_2$	forward between c_1 and c_2
	$\mid$	$\mathtt{input}\ !u\ !(d \leftarrow P_d)$	replicated server along $!u$
	$\mid$	$c \leftarrow \mathtt{copy}\ !u; P_c$	spawn a copy of $!u$ along c

Figure 2.4: Syntax for Monadic Process Expressions

usual functional constructors, with sum, product, and recursive types omitted for clarity. The primary construct of interest is $M, N ::= a \leftarrow P_{a,\overline{a_i}} \leftarrow a_1, \ldots, a_n$, which represents the internalization of process expressions via the contextual monadic construct, defining a process P that utilizes channels a_i to deliver along a.

In addition to defining processes P and Q, we observe their composition through the integration of the monadic terms M and N. The construct $a \leftarrow P_{a,\overline{a_i}} \leftarrow a_1, \ldots, a_n$ forms a process that will communicate via a newly created channel a. Subsequent operations function similarly to their initial introduction in Figure 2.2.

The construct $\mathtt{fwd}; c_1; c_2$ facilitates the forwarding of all communications between channels c_1 and c_2. This means that any message sent on c_1 is received on c_2, and vice versa.

The operation $\mathtt{input}; !u; !(d \leftarrow P_d)$ adopts a different approach compared to previous constructs. It can be conceptualized as a server awaiting requests. Upon receiving a request, it replicates itself and dispatches a copy to handle the request. This leads to the subsequent operation, $c \leftarrow \mathtt{copy}; !u; P_c$, where a request is made for a process, and we wait for it to respond with a replicated copy of the server for our use [15].

Given that the syntax presented in Figure 2.3 and Figure 2.4 is a straightforward extension of that in *session types* (Figure 2.3 and Figure 2.4 respectively), all previous

examples are applicable here and function similarly to *logical session types*.

SILL is a type-safe language that inherits the properties of deadlock-freedom, session fidelity, and communication safety, as studied in both *session types* and *logical session types*. However, it also shares the same limitations. To broaden the set of programs and the range of problems they can solve, it is necessary to relax some of these restrictions.

2.4 Manifest sharing

In the intuitionistic linear setup, processes construct a tree at runtime to ensure that each client is the sole client of a given process. This invariant is no longer maintained in *manifest sharing* due to the introduction of shared processes, which allow multiple clients to refer to a process through a shared channel. This "relaxation" introduced by *manifest sharing* enables the language to expand the range of programs that can be written [3].

To ensure session fidelity, communication over a shared channel must only occur after exclusive access to the service provided over that channel has been obtained. This is achieved by applying an acquire-release discipline to shared processes. A release relinquishes exclusive access to the process if it is available, while an acquire grants exclusive access if the rights are held. Consequently, processes may alternate between being shared and linear. For example, a successful acquisition can convert a shared process into a linear one, whereas a release can transform a linear process into a shared one [3].

To adopt this perspective of a process transitioning through phases, it is necessary to identify a process with a thread of control. In intuitionistic linear logic, a process can be associated with the channel through which it provides a service. We demonstrate the programmatic operation of the acquire-release primitives using a schematic producer-consumer scenario in Figure 2.5 [3].

```
produce : 1 ← A, queue A

c ← produce ← x, q =
      q' ← acquire q ;
      q'.enq ;
      send q' x ;
      q ← release q' ;
      c ← produce ← x, q
```

```
consume : 1 ← queue A

c ← consume ← q =
      q' ← acquire q ;
      q'.deq ;
      case q' of
      | some ← x ← recv q' ;
                   q ← release q' ;
                   c ← consume ← q
      | none ← q ← release q' ;
                   c ← consume ← q
```

Figure 2.5: The acquire-release primitives are illustrated in a producer-consumer scenario, with shared channels depicted in red and linear channels in black.

In Figure 2.5, we assume that a shared process of session type $queueA$, containing shared items x of type A, is the source of the shared channel q for both processes. To distinguish shared channels and shared session types from linear channels and session types,

which are typeset in black, we typeset shared channels and shared session types in red in the program code. Additionally, it is assumed that if the queue is empty, the session type *queueA* recurs rather than terminates upon dequeuing, which is more suitable for a producer-consumer environment [3].

In Figure 2.5, the producer and consumer processes attempt to interact with the queue by sending corresponding acquire and release messages. For instance, when the statement q' ← acquire is executed, the produce process issues q, resulting in the queue's linear channel q', which the process can use to enqueue an element if successful. The process then issues q ← release; q' to release the now linear queue process supplying along q'. This action yields the shared channel q of the queue, allowing another producer or consumer to take their turn [3].

Acquire and release represent synchronization points in inter-process communication, akin to the sending and receiving of messages. If acquire and release were introduced as the sole operational primitives, session types would no longer accurately define communication protocols. *Manifest sharing* enhances the type system by specifying the points in communication where acquire and release must occur, thereby restoring the descriptive power of session types [3].

$$
\begin{array}{lcll}
A_S & ::= & \uparrow_L^S A_L & \text{Shared type} \\
\\
A_S, B_S & ::= & \downarrow_L^S A_S & \text{input channel of type } A \text{ and continue as } B \\
& | & \ldots & \text{types present in SILL}
\end{array}
$$

The syntax presented above illustrates the necessary modifications made to SILL to incorporate manifest sharing. This begins with the introduction of a new process type, the *shared type*. Despite its significance in this implementation, the shared type has limited utility as it must be acquired before use. Conversely, *linear types* are the process types previously utilized, supporting the same types and operations, with the added capability of being shared.

Referring back to Figure 2.5, the distinction between the utilization of shared and linear types becomes evident. Whenever a producer intends to sell (share) a new object x, it must first acquire queue A before releasing the list of objects. Similarly, the consumer must follow the same acquire-release procedure for queue A to purchase an item from it.

While *manifest sharing* offers greater expressiveness compared to SILL (or other implementations of *logical session types*), it does not come without cost. Although we preserve session fidelity and communication safety, we cannot guarantee the same for the deadlock-freedom property we previously achieved.

2.5 Sharing with Differential Linear Logic

A different perspective from the one we adopted is to extend a system of logical session types by developing a correspondence between session types and the operators of differential linear logic (DiLL). A similar approach was developed by Rocha and Caires [6] to expand Linear Logic (LL) with differential constructions. This language extension occurred fairly early on in the creation of this system [5].

From a pragmatic standpoint, this methodology allows the augmentation of *SILL* incorporating a new concept, *cells*. These *cells* become the memory for data transactions, and, in them, a process can execute the usual read and write operations.

The new concept can be shared (*co-contraction*) by duplicating a cell usage or discarded (*co-weakening*). When confluent situations arise, these *cells* preserve all possible results.

Duplicating a cell usage when the *share* operator is called allows other processes to use it, while the complementary *free* will free all versions of the saved cell [5].

While this approach enables maintenance of a deadlock-free solution and slightly increases the language's expressivity, it remains somewhat restrictive. A significant constraint imposed through this approach is that *cells* can never contain other linear structures. *Manifest sharing* complements *SILL* with a wider expressivity and leaves room for future research on how to ensure deadlock-freedom.

2.6 Priority GV

Priority GV, denoted by the code [20], employs a wholly distinct approach, founding itself on the concept of *Good Variation* (GV) [21, 22]. GV is defined as a binary session typed concurrently with the λ-calculus, where precisely two processes share each channel. Binary session types ensure two essential characteristics of communication: safety and session fidelity. Deadlock freedom constitutes a third essential characteristic and ensures that processes are free from cyclic dependencies. Although deadlocks can result from interleaved sessions and are not entirely ruled out by binary session types alone, numerous additional methods have been developed to ensure deadlock freedom. These include DILL [15], Wadler's Classical Processes (CP) on which GV is based [17], and others.

The development of Priority GV as a variation of GV is predicated on the principle of decoupling channel generation from thread spawning. This approach is undertaken to facilitate the execution of cyclic-structured processes while maintaining deadlock freedom through the utilization of priorities. The development trajectory of Priority GV originates from CP to Priority CP, building upon the foundational principles established in the preceding line of work.

The fundamental concept, as implied by the eminent *Priority* keyword, is to assign a numerical priority value to each session type operation. For each operation of the program, a number greater than every other would be attributed to previous operations of the

same process. In scenarios involving interactions between processes, as observed in the aforementioned instances, the channel prioritizes the process with the highest urgency.

This approach is inferior to the DiLL-based and other logical approaches in that it lacks compositional properties. In other words, priorities must be attached to types globally in a program, rather than locally in threads or processes. Furthermore, the assignment of priorities introduces a degree of intricacy to the system, necessitating that programmers implement these priorities in a manner that is globally consistent concerning session types.

3 Language

The previous chapter provided a comprehensive analysis of the process syntax of *SILL*, examining it from a higher-level perspective and utilizing the work of Balzer [3] to thoroughly understand the mechanisms and applications of manifest sharing.

3.1 Syntax

Now we must change our focus to the practical implementation of the language. We will start by defining the syntax of the language, which is divided into two main components: types and processes. The syntax of types is defined in Figure 3.2 and the syntax of processes is defined in Figure 3.1.

Process Expressions	`P,Q`	`::=`	`a ← spawn (expression)(,); P_a`
		|	`send c x ; Q`
		|	`x:ty ← recv q; P`
		|	$x \leftarrow recv\ q;\ P$
		|	`sendc c d; Q`
		|	`x: stype ← recv c; Q`
		|	$x \leftarrow recv\ c;\ P$
		|	`case c of case_list`
		|	`c . label ; P`
		|	`wait c ; P`
		|	`close c`
		|	`fwd c_1 c_2`

Figure 3.1: Practical Syntax for Process Expressions

A good place to start our in-depth study of SILL's practical syntax would be on how

we can spawn a new process. In Figure 2.4 it is written as $a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a$, and as we said, here, we are launching a process M and communicating with it by the channel a. In a practical code example, we have something like `a ← spawn (expression); P_a` , where I use the term "something like" because we are not able to write the exact code as in the theoretical syntax without detailing the *expression*, which is the process that we are launching. This *expression* can be a previously defined function, or we can write a complete program in it, and as such it is more easily understandable in the examples below.

Our send syntax that was previously presented as $x \leftarrow$ `input` $c; Q_x$ is shown as `send c x ; Q` in the code syntax. Q_x or `Q` is simply the continuation of the process that will be executed after the input of the value `x` along the channel `c` . In our parser, we will call a `send` function with parameters `c` , `x` and the continuation of our process `Q` .

Our send counterpart, the process of receiving a value through channel c sent by another process in a parallel execution, `_` $\leftarrow$ `output` $c\ M; P$, is now represented as `x ← recv q; P` in the code syntax. The continuation of the process, `P` , will be executed after the output value read from channel `c` is written in `x` . As we would expect, the parser calls a receive function (`recv`) with parameters `c` , `x` and the continuation of our process `P` . Also, in most of our examples the variable that will receive the value read from the channel will be a brand new, as such we need to declare its type. In `x:ty ← recv q; P` we are declaring the type `ty` of the variable `x` and this field will be forwarded to the `recv` function in the parser as well.

Similarly, the syntax for sending a channel `d` along a channel `c` that was symbolized as $d \leftarrow$ `input` $c; Q_d$ is coded as `sendc c d; Q` . The syntax is once again very similar to the previous one; the key elements stay the same with, the exception of the continuation of the process simplified from Q_d to `Q` , only the keyword chosen is different in an easier and more intuitive way to write and read the program.

As for the `sendc` or the *sendchannel* counterpart, we now have a `recvc` (*receivechannel*) function. It was referred to as `_` $\leftarrow$ `output` $c\ (d \leftarrow P_d); Q$ and its practical use is done as `x ← recv c; Q` . This example appears to be a bit more complex but isolating its key elements we rapidly find the similarities. We can easily identify the channel `c` that is being read from, the variable `x` that appears only in the coding syntax since previously we weren't to worried about where the channel `d` was going to be stored, now, however, it makes sense for us to specify it since in real applications we need to access it later.

On this note, `x` , can also be a brand new variable, and for that, we need to declare its type (`stype`), as we can see in the syntax `x: stype ← recv c; Q` . We will analyse `stypes` more in depth when we analyse Figure 3.2. Regarding the channel received in `recvc` , `d` , it is the point of connection between the current process `Q` and the parallel process `P` . Finally, and equal in both syntaxes, we have our current process `Q` , more specifically, symbolizing a non-optional continuation.

According to Figure 3.1, next on our list comes the syntax for the *case* function. In

Figure 2.4 we had `case` c `of` $\overline{l_j \Rightarrow P_j}$. There is a channel c that serves as a mean of communication between the two processes. We have the *case* and *of* keywords that are there to identify the function easily and to write our program. And finally we have $\overline{l_j \Rightarrow P_j}$, that symbolizes our list of possibilities, each l_j behaving as P_j. In a more practical approach, we need to specify each and every possible choice as well as its type. With this in mind, we understand the need of the *case_list* syntax that used in `case c of case_list` . The `case_list` is composed of a minimum of one `label ( x )` that makes our channel `c` behave according to `x` 's `stype` (`case_list ::= x : stype, case_list`).

Also worth noting is that the continuation of the process is not done after our choice is presented, but instead it is presented as an option itself. In order for our process to be specified correctly, we need one of the options to close it, and as such, it makes sense that our `case_list` has a minimum of one element, since if there is an option to present, there is at least one channel that still needs to be closed.

In contrast, $\_ \leftarrow c.l_j; P$, is easy enough to apply in practice. `c . label ; P` keeps the same channel `c` , the process is forced to continue along as `P` , and the label l_j is the choice that we are making; it is the variable `x` that is defined along the `case_list` .

The following function in our code syntax is the *wait* function. Previously written as $\_ \leftarrow$ `wait` $c; P$ it is now simplified to `wait c ; P` . Basically, we wait for a process `Q` (omitted in the syntax, which is the process spawned alongside the creation of the channel `c`) connected to us, process `P` , to close the channel `c` through the `close` function. Moreover, afterwards, we can continue our process as `P` .

Referred to by `close` c in Figure 2.4, the *close* function is now written as `close c` in the code syntax. It is a simple function that closes the channel `c` and terminates the process. It is one of the two ways to terminate a process, having no continuation.

The other way to terminate a process is through the $forward$ function. Just as with *close*, we must not write a continuation. Different from *close*, however, $forward$ does not close the channel; it simply passes through what it receives. In Figure 2.4 we had `fwd` c_1 c_2, and it remains unchanged in the code syntax (`fwd c_1 c_2`), applying the explanation to our specification `fwd` will make forward of what we receive in `c_1` to channel `c_2` .

When writing our programs, we need to prepare the *typechecker* for the types it should expect. For this purpose, we need *stypes*.

The types in Figure 3.2 describe, in the linear setting, the protocol followed by each endpoint of a channel. Each constructor specifies an exact communication action, followed by the continuation of the session, directly matching the constructs from Figure 3.1.

The case `ty lst` represents sending a value of type `ty` and then continuing according to `st`. This matches the `send` construct, where the type specifies the payload to be transmitted before the protocol proceeds.

```
Stype  lst  ::=  ty ^ lst
             |   ty  =>  lst
             |   (lst)  *  lst
             |   (lst)  -o  lst
             |   & (choice_list)
             |   + (choice_list)
             |   @
             |   v
             |   rec  v . st
```

Figure 3.2: stype Syntax

Similarly, `ty => lst` represents receiving a value of type `ty`, after which the protocol continues as `lst`. This corresponds to the `recv` construct, enforcing that the received value has the expected type.

The type `(lst) * st` represents sending a fresh channel whose protocol is `lst`, and then continuing as another `lst`. This matches the `sendc` construct, where channel passing is used to delegate part of the protocol to another process.

The dual case `(lst) -o lst` represents receiving a fresh channel with protocol `lst`, then continuing according to `lst`. This corresponds to the channel-receiving variant of `recv`.

The type `&(choice_list)` represents offering a set of labeled branches to the peer, waiting for it to select one. This directly corresponds to the `case` construct, where the process must be ready to execute the chosen branch.

The type `+(choice_list)` represents selecting one of the labels offered by the peer. This matches the `select` construct (`c.label`), allowing the process to steer the protocol in a chosen direction.

The type @ marks the termination of the session. At this point, no further communication is possible, just as with the `close` and `wait` constructs in the process syntax.

Finally, `v` and `rec v. lst` represent type variables and recursive session types, respectively, enabling the definition of protocols with repeated or infinite behavior. Recursion is essential to express loops or unbounded exchanges within the session.

Conclusion. In this section, we have presented the practical syntax of our process expressions and the corresponding linear session types (*stypes*). Each construct of the process language has a precise typing in the linear setting, specifying exactly the protocol followed along each channel. This formalization establishes a foundation for safe, structured communication in concurrent programs: sending, receiving, channel delegation, choice, and termination are all rigorously captured by the types.

Having established this linear core, we are now prepared to extend the language to support shared channels. The next chapter introduces the notion of *manifest sharing* [3], showing how processes can safely share communication channels while preserving type soundness. This extension will build directly on the linear base we have defined, reusing and generalizing the concepts of process syntax and session types.

3.2 Manifest Sharing Interpretation

Up to this point, our discussion has focused exclusively on *linear stypes*, where each channel endpoint is owned by exactly one process and must be used according to its protocol without duplication or loss. We now extend this model to incorporate *shared stypes*, following the acquire-release discipline of *manifest sharing* [3] introduced in chapter 2. In this setting, a channel may be shared among multiple clients, alternating between a shared phase, where it can be referenced concurrently, and a linear phase, where a single client has exclusive access to the service it provides.

The syntax in Figure 3.3 extends the notion of stypes by introducing two distinct categories: *linear stypes* (`lst`) and *shared stypes* (`sst`). While the previous chapter presented purely linear stypes, here the linear fragment has been refined to integrate with the acquire-release discipline and to support more expressive interaction patterns. These "upgrades" enable linear channels to interoperate seamlessly with shared channels, allowing for smooth transitions between phases without compromising session fidelity.

Stypes	`sts`	`::=`	`lst`	linear stype
		|	`sst`	shared stype

Changes in the old linear stype syntax:

linear_stype	`lst`	`::=`	`ty ^ stypes`	Sending ty type
		|	`ty => stypes`	Receiving ty type
		|	`( stypes ) * stypes`	Sending channel type
		|	`( stypes ) -o stypes`	Receiving channel type
		|	`sh lst`	Receiving channel type
		|	...	

Shared stype Syntax:

shared_stype	`sst`	`::=`	`rq lst`	request stype
		|	`(sh) v`	variable
		|	`rec v . sst`	Recursive type

Figure 3.3: Stypes Syntax

A `lst` describes a protocol that applies when a channel is in its *linear phase*, that is, when a single client has exclusive ownership of the channel endpoint. This refined definition includes all the standard types of the purely linear setting (send, receive, choice, etc.). Also, it incorporates the necessary constructs to hand over control back to the shared phase. We will see all the changes and new expressions below.

An `sst` describes the protocol governing a channel in its *shared phase*. In this phase, the channel can be referenced by multiple clients, but no direct communication is allowed until a client explicitly *acquires* the channel. Acquisition grants exclusive ownership, transitioning the protocol into the associated `lst`. Conversely, a *release* operation returns the channel to the shared phase, making it available for other clients to acquire.

This separation between `lst` and `sst` provides a clean formal foundation for reasoning about channels that alternate between exclusive and shared use, a key feature of *manifest sharing*. The following sections detail the syntax and typing of each category, showing how the acquire-release discipline is embedded in the type system.

As introduced in Figure 3.3, our session types are now divided into two categories: linear stypes (`lst`) and shared stypes (`sst`). We begin with the linear fragment. While the familiar constructs of the purely linear setting are preserved, their continuations now range over all stypes, allowing smooth transitions between the linear and shared phases. Additionally, a new construct is introduced to return a channel to the shared phase explicitly.

The case `ty ŝtypes` describes the sending of a value of type `ty`, after which the protocol continues as `stypes`.

Dually, `ty => stypes` specifies the reception of a value of type `ty`, continuing as `stypes`.

The construct `(stypes) * stypes` represents the sending of a fresh channel of type `stypes`, after which the protocol proceeds as $stypes_0$.

Symmetrically, `(stypes) -o stypes` represents the reception of a fresh channel of type `stypes`, before continuing as `stypes`.

These constructs were modified to support the interleaving of linear and shared phases, although not all were considered initially. Even the `stypes` encapsulation, despite being considered, was not the first approach. Through some trial and error during the implementation and completion of the typechecker, this design was chosen as the most appropriate approach, not only for the sake of implementation and debugging, but more importantly for ensuring the long-term scalability of the project.

Finally, the new construct `sh lst` allows a process to *release* exclusive ownership of a channel, making it available again as a shared resource. This marks the point where a linear channel transitions back into the shared phase, in accordance with the *acquire-release* discipline of *manifest sharing* [3].

The syntax of shared session types is also shown in Figure 3.3. An `sst` governs the behaviour of a channel while it is in the shared phase. In this phase, multiple clients may

hold references to the channel, but direct communication is only possible once a client explicitly acquires it.

The construct `rq lst` denotes such an acquisition request. A successful request transitions the channel into the linear phase, where it behaves according to the linear protocol `lst`.

As with linear types, recursion is supported through variables `v` and recursive definitions `rec v . sst`, and given the nature of our shared processes, this `sst` becomes essential. Shared processes can be indefinitely reused, and since we do not know how many times a channel will be `requested` , we cannot let our channels close. This will be further explained in the future, but since we cannot close a shared channel, even if our parser allows it in the linear phase, our typechecker won't, and declaring it as recursive becomes the only way to declare its type correctly.

In this way, shared stypes provide the minimal structure needed to capture the acquire-release discipline: they specify how a channel can be acquired and how its shared availability may persist through recursion.

Extended Syntax for Linear Process Expressions:

```
P,Q  ::=  a ← sspawn (expression)(x̄, ū); P
      |   request c; P
      |   share c; P
      |   closeConnection c; SP
```

Extended Syntax for Shared Process Expressions:

```
SP  ::=  a ← sspawn (expression)(x̄, ū); SP
     |   openConnection c;  P
     |   sfwd c_1 c_2
```

Figure 3.4: Extended Syntax for Processes Expressions

The grammar in Figure 3.4 shows the expansion applied to the previous process syntax. The new constructs enable the creation (`sspawn`) and interaction with shared channels. This interaction has two parallel sides, and the client's occurs exclusively in linear processes (which makes sense since the one that can receive multiple requests is the server, not the client) through the `request` and `share` constructs. At the same time, the server-side is only halfway present in this syntax with its `closeConnection` construct.

Shared Spawn:

`a ← sspawn (expression)(x̄, ū); P`

This expression spawns a new process that executes according to the protocol `expression` along a brand new channel `a` (of type *shared stype* and also in agreement with the

protocol of `expression`). Along with other parameters for `expression`, we pass along *overlinex* and $\overline{u}$, linear and shared channels, respectively. In most of our examples, these parameters will be empty `(,)`, but especially when we want to provide a shared channel through multiple other processes, passing it along here becomes simpler and easier to read.

Request:

```
request c; P
```

A `request` is used by a client process before continuing its execution `P` to *acquire* exclusive rights to a shared channel `c` from which it can then communicate linearly. From the moment the request is accepted, the execution proceeds as `P`, and before a `share` is called, we have the guarantee that no other process can interfere with our communication. On the other hand, this also means that our typechecker will need to guarantee that a `share` is eventually called for every `request` made, otherwise we would create a deadlock situation (there are, in fact, other possible deadlocks, but at least this one is possible to detect and avoid before runtime).

Share:

```
share c; P
```

The counterpart to our `request` construct is present in this expression. A `shared` of the previously `requested` channel `c` frees it from exclusive ownership and allows other clients to acquire it (and unlock their execution). After the `share` is called, our process continues linearly as `P`.

Close Connection:

```
closeConnection c; SP
```

Finally, the last expression is part of an incomplete pair. A `closeConnection` occurs concurrently with a `free`, which also means that it will only occur when the client allows it, and returns our (server) process to its previous shared state. For the sake of efficiency, we need to be cautious and avoid allowing our clients to perform secondary and unrelated tasks. At the same time, a connection is open, as this would result in a delay to the overall execution. On the other hand, if the client needs to perform other tasks before any other process can interact with the server, it can block other interactions by delaying the `share` of the channel and consequently the `closeConnection`.

Alongside our types in the previous figures, the linear process grammar provides the operational discipline that matches the type-level distinction between linear and shared types. Although correctly used, they ensure that channels can safely alternate between exclusive and shared use, introducing a new level of complexity while writing our programs. A simple example of two clients trying to access two different servers in a different order

can lead to troublesome executions, where each one successfully connects with a server but has its execution blocked while waiting for the other client to release the other server. This apparently simple example can lead to a deadlock situation that we must be careful to avoid. In a small project, establishing a priority order for the servers to be called may be straightforward, but in a larger project, this may not be feasible.

Also, 3.4 presents the extended syntax for shared processes. While linear processes are guaranteed to use their channels only once at a time, the shared counterpart can be accessed by multiple clients (`requests`) concurrently. For this reason, our grammar introduces the minimum expressions necessary to manage shared channels safely and restrains the main communication to the linear phase of the channel.

Shared Spawn:

$$\texttt{a} \leftarrow \texttt{sspawn (expression)}(\overline{x}, \overline{u})\texttt{;} \quad \texttt{SP}$$

As the first construct in our grammar, `sspawn` is analogous to the linear `spawn` or `sspawn`, but for shared processes which can only spawn other shared processes and never linear ones. By this construct, we create a new process tasked with obeying the protocol defined in `expression` along a fresh channel `a` of type shared. As arguments (possibly empty), we will pass the linear channels in $\overline{x}$ and shared channels in $\overline{u}$ according to the declaration of `M`. Lastly, and again similar to previous examples, we have the continuation of our process `SP` that will be executed in parallel with the spawned process `expression`.

Open Connection:

$$\texttt{openConnection c; P}$$

Since shared channels cannot be used directly for communication, these channels must be *acquired* from the client side. On the shared process (which we can view as a server), we must *accept* or `open` the connection attempt from such clients, guaranteeing that only one `request` is accepted for each texttt openConnection. That is the use of this construct, the first interaction our shared processes always do is the *opening* of a connection from which on they will behave exactly like a linear process `P`.

Shared Forward:

$$\texttt{sfwd c\_1 c\_2}$$

This construct forwards communication between two shared channels `c_1` and `c_2`, making them indistinguishable for subsequent interactions, and will be one of the most essential process expressions in our shared examples. Our shared processes cannot be closed; our typechecker will have an extra layer of complexity to ensure that shared processes are well-typed. We will discuss this further later. Since shared channels must not be closed and our processes must always end one execution and wait for another process to connect and initialize a new one, the `sfwd` construct will be used to forward the requests of one shared process that has already completed its cycle to a freshly spawned one.

Together, the expansion of the linear process syntax and the shared process syntax implement the primitives for *spawning*, *acquiring*, and *sharing* shared processes along with the means for these processes to correctly manage themselves (*opening* and *closing* connections as well as *forwarding* requests).

3.2.1 Coin flipper

With the syntax established, we can now proceed to a practical program that is only possible with our shared extension. One example presented by *F. Pfenning* [3] is illustrated below.

$$coin = \uparrow_L^S \oplus \{\overline{head :\downarrow_L^S coin}, \overline{tail :\downarrow_L^S coin}\} \qquad coin_{head} : \{coin\}$$

$$\begin{aligned} &c \leftarrow coin_{head} = \\ &\quad c' \leftarrow accept\ c; \\ &\quad c'.head; \\ &\quad c \leftarrow detach\ c'; \\ &\quad c \leftarrow coin_{tail} \end{aligned}$$

This example starts with the declaration of the type coin in parallel with a process that will behave according to such a protocol. Ideally, we would have a process $coin_{head}$ that would behave as described and, at the end, call a $coin_{tail}$ that, as expected, would change the label of our channel from *head* to *tail*. Ultimately, it would call a new $coin_{head}$, completing the cycle and providing a complete coin, which can be turned indefinitely.

I say 'ideally' because, due to other limitations with our compiler, this will not be the final version we will submit to our typechecker. However, we can, in fact, write this program using our grammar as illustrated in Figure 3.5. The $coin_{head}$ must start by *accepting* a connection with a client through c from which they start communicating linearly through c'. Despite Figure 3.5 not specifying a c' when the `openConnection` is called, internally, a new channel is in fact created since the original c will be blocked by other clients' `requests`. This internal architecture will also guarantee that the communication between a client and a server, once established, is exclusive and unhindered by other processes or clients.

Lastly, after linear communication is concluded through the *detach* (or `closeConnection`) operation, the linear channels are closed, and we pass on the pending clients to a freshly spawned process $coin_{tail}$. Once again, Figure 3.5 works very similarly, but we differentiate the `spawn` of the new process from the actual *forwarding* (`sfwd`).

For this to be a complete program, we need the processes to consume our coin. Those are nd_choice and $coin\_flipper$.

```
sh stype Coin  rec  x. rq  +{head: sh (sh) x, tail: sh (sh) x};

coin_head : unit → {Coin}
fun u -> c ← {
          openConnection c;
          c.head;
          closeConnection c;
          d ← sspawn (coin_tail ()) (,);
          sfwd d c
      }
end;

coin_tail : unit → {Coin}
fun u -> c ← {
          openConnection c;
          c.tail;
          closeConnection c;
          d ← sspawn (coin_head ()) (,);
          sfwd d c
      }
end;
```

Figure 3.5: Rough translation of *manifest sharing's* [3] coin example (part one).

$$
\begin{array}{l}
nd\_choice : \ \{\oplus\{\overline{yes : 1}, \overline{no : 1}\}\} \\
\quad d \leftarrow nd\_choice = \\
\quad\quad c \leftarrow coin_{head}; \\
\quad\quad f \leftarrow coin\_flipper \leftarrow c; \\
\quad\quad c' \leftarrow acquire\ c; \\
\quad\quad case\ c'\ of \\
\quad\quad \mid head \rightarrow c \leftarrow release\ c';\ d.yes;\ wait\ f;\ close\ d \\
\quad\quad \mid tail \rightarrow c \leftarrow release\ c';\ d.no;\ wait\ f;\ close\ d
\end{array}
$$

$$
\begin{array}{l}
coin\_flipper : \ \{1 \leftarrow coin\} \\
\quad d \leftarrow coin\_flipper \leftarrow c = \\
\quad\quad c' \leftarrow acquire\ c; \\
\quad\quad case\ c'\ of \\
\quad\quad \mid head \rightarrow c \leftarrow release\ c';\ close\ d \\
\quad\quad \mid tail \rightarrow c \leftarrow release\ c';\ close\ d
\end{array}
$$

The process running $coin_flipper$ will, as the name implies, flip the *coin*. In other words, it will read the value of our *coin* and ignore it, so that the next time the value is read, it returns the opposite value. This works on the premise that when the $coin_{head}$ is consumed or executed, it calls a $coin_{tail}$, which is "flipped".

```
coin_flipper : unit → {@ ← ; c:(sh)Coin}
fun u -> d ← {
          request c;
          case c of
              head: (
                  share c;
                  close d
              )
              tail: (
                  share c;
                  close d
              )
      }
end;

nd_choice : unit → {&{yes:@, no:@}}
fun u -> d ← {
          c ← sspawn (coin_head ()) (,);
          f ← spawn (coin_flipper ()) (,c);
          request c;
          case c of
              head: (
                  share c;
                  d.yes;
                  wait f;
                  close d
              )
              tail: (
                  share c;
                  d.no;
                  wait f;
                  close d
              )
      }
end;
```

Figure 3.6: Rough adaptation of *manifest sharing's* [3] coin example (part two).

This is the outline of what happens, but we need to detail a few steps while writing the actual program. `coin_flipper` starts by making a `request` to *acquire* access to coin. After being granted exclusive rights, we read the state of coin by calling `case` of our linear channel c'. Since we want to flip coin, we actually take the same action independently of

the coin status, we *free* it from our grasp (*release*) and end our activity (`close`).

For our program to be useful, we need to actually be able to return a value or access the random status of our coin. This role is played by the process running *nd_choice*. It will initiate a coin, spawning a $coin_{head}$ followed by a *coin_flipper* to which we share *c*, the shared channel to coin.

With the groundwork done, `nd_choice` moves on to `request` the right to *acquire* the state of the coin. However, we understand that this happens simultaneously to `coin_flipper`, who attempts to do the same. As a result, our `nd_choice` can sometimes answer with `yes` and other times with `no` depending on who gets its request fulfilled first.

Finally, it waits for the `coin_flipper` to close and ends its execution.

Figure 3.6 shows the rough or literal translation to our grammar. The final element missing for a complete working program is a process to call our `nd_choice` and take the appropriate action depending on its result.

```
;; m <- {
          c ← spawn (nd_choice ()) (,);
          case c of
              yes: (
                  print 1;
                  wait c;
                  close m
              )
              no: (
                  print 0;
                  wait c;
                  close m
              )
 };;
```

Figure 3.7: Main program for the nondeterministic `Coin` example, branching on `yes` or `no` outcomes.

Since our program is being used exclusively as a demonstration for our coin, `nd_choice` can be called directly from our main (as in Figure 3.7), and as a result, we will just `print 1` for `yes` and `0` for `no`.

Conclusion. This section thoroughly explained the expansion made upon the linear implementation of SILL. Introducing shared types and the function mechanisms to manage them gives our language a lot more freedom when writing programs, but such benefits do not come freely. The added functionality of operations that enable the modeling of client-server architectures and allow multiple clients to interact with the same resource also brings the risk of new deadlocks. It requires more attention while writing more complex

programs. In any case, despite the possible exceptions that the *typechecker* allows, it avoids much trouble before runtime, and this subject will be delved into deeper in the following section.

3.3 Typechecker

After correctly implementing support for the new operations, it was time to complete the typechecker itself.

`let decls, e, ty = Typechecker.check_program desugared_prog in`

Given our *desugared* program, which means that syntactic sugar has been removed, we are left with a more basic version of our program that is easier to manipulate and can be converted into our internal types. Both steps were crucial to ensure that the typechecker would receive well-structured input and process it correctly, thereby avoiding unnecessary complications and allowing us to focus on the core logic of the typechecking itself, rather than dealing with parsing errors or syntactic ambiguities. We need to ensure that it is well typed.

Our programs always have a main expression for which we need to infer its type. However, for other expressions, such as function definitions, we need to have their types declared beforehand. Let us start by analysing simple programs and then move on to more complex ones.

```
;; c ← {
     d ← spawn {
         v1:int ← recv d;
         send d v1;
         close d
   } (,);
     send d 1;
     v1:int ← recv d;
     print 0;
     wait d;
     close c
};;
```

Figure 3.8: Simple linear example of SILL

In Figure 3.8 we have a simple linear program that spawns a process that receives an integer from a channel `d`, sends it back, and closes the channel. The main process sends the integer `1` to `d`, receives it back, prints `0`, waits for the spawned process to close `d`, and finally closes its own channel `c`.

As we can see, the main expression starts with `;; c ← { ...`, then we spawn a process with `d ← spawn { ... } ( , )`, and lastly we interact with the spawned process through `d` without declaring which types are expected in each step. This means that, for process interactions to be *typesafe*, the typechecker will need to infer the types of

all channels involved. In this case, it will first infer the type of the spawned process, and then use it to infer the type of the main expression.

Now, let us rewrite the same example, but this time, we will extract the spawned process into a function definition with an explicit type declaration. In a small program, there would be no need for this; our typechecker is powerful enough to infer all necessary types. However, in larger programs, besides avoiding initial mistakes, this is a common practice to improve readability and maintainability.

```
pass : unit → { int ⇒ int ^ @ }
fun u -> c ← {
          v1:int ← recv c;
          send c v1;
          close c
    }
 end;

;; c <- {
      d ← spawn (pass ()) (,);
      send d 1;
      v1:int ← recv d;
      print 0;
      wait d;
      close c
};;
```

Figure 3.9: Rewritten simple linear example of SILL

As we can see, the execution flow of the program remains unchanged in Figure 3.9. Our *main* executable code remains the same, but now we have a function definition with an explicit type declaration for the spawned process. This means that the typechecker will first verify whether the function definition is well-typed, eliminating the need to infer its type, and then use its type to infer the type of the main expression.

Let us analyse it line by line. The function definition starts with `pass : unit → { int → int ^ @ }`, which declares that the function *pass* will receive no arguments (`unit`) and will return a channel of type `{ int → int ^ @ }`. The following line, `fun u → c ← { ...`, defines the function itself, where u is the empty argument of type `unit` and `c` is the channel that will be used for communication. From here on, inside the function body, we will decapsulate `c` according to its type. The first action is `v1:int ← recv c;`, which receives an integer from `c` and binds it to `v1`. This means that the channel `c` on our end just received an integer `int =>` and shed the first layer of its type. From the following line onwards, the expected type for our channel c will be `int ^ @`. The following line, `send c v1;`, sends back the integer `v1`. This interaction with our

channel is of type `int ^`, so we are left with `@` for the type of `c`. Finally, we have `close c`, which closes the channel `c`, which is of type `@`, and the function definition ends. Having successfully typechecked the function definition, the typechecker "accepts" that `pass` is well typed and moves on to the main expression.

In the main expression, the first line is similar to a function declaration, with good cause; after all, our main function is precisely a function that runs in a dedicated process. This process will indicate closure upon the channel c exactly like any other function would, the difference would be the receiver that is no longer another parent function of our program, but instead, it declares the end of the program itself `;; c ← { ... } ;;` .

The line `d ← spawn (pass ()) (,);` spawns a new process that runs the function `pass`. As we have seen, this function is spawned with no arguments (`pass ()`) and, despite not having mentioned the subject before, the `pass` function also receives no other channels to communicate with `(,)`. It is important to note that despite the communication between our `main` (process) and our `pass` (process) occurring through a single channel, and both ends needing coherent communication (our program would not be well typed if both processes could try to send information at the same type), our `pass` function is calling this channel as `c` and our main calls `d` to the other end of the same channel, this is possible since they are simply labels and these labels are names contained inside the process and work independently.

With the groundwork laid out, we can now focus on the actual code for this process. The typechecker already knows that the other end of the channel is awaiting to receive an integer, so our first interaction with `d` is of type `int ⇒`. Our code line corresponds exactly to this expectation, `send d 1;` sends the integer 1 to the channel d, which is coherent with the type of the other end of the channel. Likewise, the next line `v1:int ← recv d;` receives an integer from `d` (`int ^`), which is also coherent with the type of the other end of the channel, which is now expecting to send an integer `int ^`. Furthermore, finally, the last line of our main that interacts with `d` is `wait d;`, once again, the coherent counterpart to the closing of the channel `c` in the `pass` function.

We purposefully chose to ignore the `print 0;` line in our analysis, as it does not interact with any channels and does not influence the typechecking of our program. The last line `close c` is in fact relevant, and when typing our `main` function would be the only line "visible" since it would be the only line that interacts with the channel `c`, the outside of the function scope but its type, `@`, is inferred.

This concludes the type checking of our program; it begins by verifying the `pass` function definition. Once it is confirmed to be well-typed, it proceeds to the main expression and infers the counterpart of the type of `pass` for the channel `d` through the duality of session types, verifying our interaction with this process through channel `d`. Finally, it infers the type of `c` as `@` and verifies our interaction with it.

With this understanding, we can achieve an even more precise separation of responsibilities within our program. We can define all function types at the very beginning of our program file, followed by the function definitions themselves. Finally, the main expression

```
stype PassType  int ⇒ int ^ @;
stype DelegateType  @;

pass : unit → {PassType}
fun u -> c ← {
          v1:int ← recv c;
          send c  v1;
          close c
    }
end;

delegate : unit → {DelegateType}
fun u -> c ← {
          d ← spawn (pass ()) (,);
          send d  1;
          v1:int ← recv d;
          print  0;
          wait d;
          close c
    }
end;

;; c <- {
      d ← spawn (delegate ()) (,);
      wait d;
      close c
};;
```

Figure 3.10: Extended example in SILL with separated `PassType` and `DelegateType`

can be written in the simplest form possible, creating a process that performs all the work previously done in the main function itself and waits for it to finish before closing the program.

The Figure 3.10 shows this exact structure. At the top of our program, we have two stype declarations: `stype PassType int ⇒ int ^ @;` and `stype DelegateType @;` . These declarations define the expected communication protocols for the functions `pass` and `delegate`, respectively. Following the stype declarations, we have the function definitions. The `pass` function remains unchanged from our previous example, while the `delegate` function encapsulates all the logic that was previously in the main expression. It spawns a process running `pass`, interacts with it through channel `d` following the same protocol, and finally closes its own channel `c`.

At last, our main expression is present in its simplest form. It delegates its previous work by spawning a process `delegate` connected through channel `d` and waiting for it to finish before closing the program with `close c`.

3.3.1 Manifest Sharing Rules

Before moving on to a shared example, we should further analyse the rules behind the behaviour of the new operations.

$$\frac{\Gamma,\ x_S :\uparrow^S_L A_L\ ;\ \Delta,\ x_L : A_L\ \vdash_\Sigma\ Q_{x_L} :: (z_L : C_L)}{\Gamma,\ x_S :\uparrow^S_L A_L\ ;\ \Delta\ \vdash_\Sigma\ x_L \leftarrow \mathsf{request}\ x_S\ ;\ Q_{x_L} :: (z_L : C_L)}\ \mathrm{T}\uparrow^S_L\mathrm{LL}$$

$$\frac{\Gamma\ ;\ \cdot\ \vdash_\Sigma\ P_{x_L} :: (x_L : A_L)}{\Gamma\ \vdash_\Sigma\ x_L \leftarrow \mathsf{openConnection}\ x_S\ ;\ P_{x_L} :: (x_S :\uparrow^S_L A_L)}\ \mathrm{T}\uparrow^S_L\mathrm{LR}$$

$$\frac{\Gamma,\ x_S : A_S\ ;\ \Delta\ \vdash_\Sigma\ Q_{x_S} :: (z_L : C_L)}{\Gamma\ ;\ \Delta,\ x_L :\downarrow^S_L A_S\ \vdash_\Sigma\ x_S \leftarrow \mathsf{share}\ x_L\ ;\ Q_{x_S} :: (z_L : C_L)}\ \mathrm{T}\downarrow^S_L\mathrm{LL}$$

$$\frac{\Gamma\ \vdash_\Sigma\ P_{x_S} :: (x_S : A_S)}{\Gamma\ ;\ \cdot\ \vdash_\Sigma\ x_S \leftarrow \mathsf{closeConnection}\ x_L\ ;\ P_{x_S} :: (x_L :\downarrow^S_L A_S)}\ \mathrm{T}\downarrow^S_L\mathrm{LR}$$

Figure 3.11: Typing rules for *request/openConnection* and *share/closeConnection*.

The first rule in Figure 3.11 is related to the *request* operation. This operation, used by a client on a shared channel x_S, will provide the linear channel x_L. x_S will remain available to the process but inside Q_{x_L} communications will flow through x_L (until severing the connection with *share*).

On the server side, *request* synchronizes with *openConnection*. This operation transforms the shared channel x_S into a linear channel x_L, enabling communication over this channel.

The last two rules will basically undo what was done in the previous ones. When the linear protocol finishes, a client must release its linear channel x_L, losing access to it, and use x_S instead when interacting with the shared process further.

On the server side, *closeConnection* will synchronize with the previous *share* and transform the channel back to its shared state x_S.

As we will see in the future, this "transformation" occurs in the final Go program by using an entirely different channel. Moreover, losing its shared state means a shared process will not have access to this channel until it returns to its initial state.

3.3.2 Coin flipper

Now that the process is understood, let us delve into the typechecking of shared processes in a linear and straightforward program. Although there are several linear examples that have not been covered yet, they are not the primary focus of this work. The linear features were already developed, so the focus was on maintaining correct functionality while adding new features and making changes only when necessary. The execution was not always this simple, but that was the idea.

As referenced before (in subsection 3.2.1), it was not possible to recreate the same program as the one written. Our project, or compiler, will take the code written and translate it into a state machine in *Go*, which is easier to do and verify by reading the program in a specific order. In Figure 3.5, we declare the function `coin_head` first; however, before ending the function, we try to call `coin_tail`. Moreover, since `coin_tail` also calls the `coin_head`, we cannot verify one as valid without the other, and the compiler does not support such awareness. To support this example, more extensive work would be required, whereas we can find other examples that demonstrate the shared functionality just as seamlessly.

From the possible variations, I selected to treat the Coin like a counter. Analysing our Coin again in Figure 3.5 and seeing its usage in Figure 3.6. Our Coin only has two states, `head` or `tail`, and despite the possibility of endlessly turning the Coin, our example tries to turn it once (at the same time it tries to read its value). Then once the value is read, we convert it to `0` or `1`.

By interpreting our problem differently, we can reach a similar result. Even with the difference in execution, for the sake of parallelism, the previous names for the functions and types are retained.

```
sh stype Coin rec x. rq int ^ sh (sh)x;

coin_head : int → {Coin}
fun n → c ← {
          openConnection c;
          send c n;
          closeConnection c;
          d ← sspawn (coin_head (n+1)) (,);
          sfwd c d
    }
end;
```

Figure 3.12: Type Coin for the shared coin example.

The previous Coin type presented in Figure 3.5 has its new adaptation present in Figure 3.12, and the first half of our type Coin remains, as expected, unchanged, `sh stype`

`Coin rec x. rq`. We declare a shared type (`sh stype`) named `Coin`. Since shared types cannot be closed and must return to their initial state after usage, it is declared as recursive (labeled as `x`).

The first operation is also aligned with the expectations; shared processes need to be *requested* (`rq`) before initiating the core communication.

Both ends of our type `Coin` in Figure 3.5 end with `sh (sh) x` (just like in Figure 3.12). We get our linear channel severed and return to our original state `(sh) x`.

Seeing both the start and the end of our `Coin` being equal, we understand that the only difference is the "return" or the way our `Coin` shares its state. While in Figure 3.5 we had a labeled choice. Changing our perspective allowed for the direct return of an integer.

The similarities between the old and the new `coin_head` functions are also quite evident. Let us start with the new function and leave the analogy with the previous one for later. Our `coin_head` is defined as a function that takes an `int` and provides a channel that follows the `Coin` protocol.

Our function will remember its integer parameter as `n` and communicate through the channel `c`. The typechecker already knows the type of `c` and expects it to accept a connection, send an integer, close the connection, and somehow, after doing all that, return to the initial shared state so that other clients can interact with it. The first code line of `coin_head` is `openConnection c`, according to the protocol, which lowers the function from the shared state and `accepts` a connection with a client. The initial `rq` is satisfied, and from the following line onwards `c` is expected to be `int ŝh (sh) x`. Similarly, `int ^` is satisfied when we `send n` linearly to our client through `c`. The connection is then closed (`closeConnection c`) in line with `sh`, and what remains is to return our channel to its previous state.

Looking back, the only differences from the older version of `coin_head` are the parameter we receive, we receive `n` and declare it in the function header of the new version, and the line where we share our result, `c.head` is replaced with `send c n`. The next step is to create a process running `coin_tail` and pass future `requests` to it. Since the type provided in the function header of `coin_tail` is the same as `coin_head`, it is equivalent to say we returned to the initial shared state `(sh) x`. The new `coin_head` has no `coin_tail`, so it must call another process running a brand new `coin_head` with a different initial value $n + 1$. When `sfwd` is invoked in the new process, we successfully return `c` to its initial state.

Moving on to the processes that act as clients and interact with `coin_head` as a server, we identify two of them. In Figure 3.13, we present the functions that each of these clients will execute. The `coin_flipper` function is declared as one that takes no arguments (`unit`). Its type is `@`, meaning it only establishes an external connection and then terminates its execution. The construct `← ; c: (sh)Coin` specifies that the function requires (←) a channel `c` of type `(sh)Coin` for its execution. This channel will be used to communicate with `coin_head` as `c`. The following line defines the protocol, labelling the empty argument as u and the external communication channel as d, where only the closing

```
coin_flipper : unit → {@ ← ; c: (sh)Coin}
fun u → d ← {
          request c;
          v1:int ← recv c;
          share c;
          close d
    }
end;

nd_choice : unit → {int ^ @}
fun u → d ← {
          c ← sspawn (coin_head 0) (,);
          f ← spawn (coin_flipper ()) (,c);

          request c;
          v1:int ← recv c;
          share c;

          send d v1;

          wait f;
          close d
    }
end;
```

Figure 3.13: Racing functions for the shared coin example.

action `@` is expected. The function begins with an interaction between itself, the channel `c`, and the typechecker, which determines the behavior expected from `c` according to its declared type. Thus, the first operation must be the dual of `openConnection`, namely `rq`. The command `request c` is precisely that operator. Afterwards, the function receives the integer transmitted as the state of `coin`. Although this value is not used, it must still be read, since it forms part of the channel protocol. The final interaction with `c` is `share c`, which operates in duality with `coin_head`'s `closeConnection`. Finally, having "flipped" the `Coin`, the function terminates with `close d`, consistent with its declared type `@`.

The other process that competes for access to `Coin` is `nd_choice`. This function has neither value nor channel arguments. Its type is `int ^ @`, which indicates that it must send an integer and then close itself through the channel `d`. By executing `c ← sspawn (coin_head 0) (,)`, a shared `coin` is spawned together with the shared channel `c`. Once the `Coin` has been spawned, its channel is passed as an argument when spawning a `coin_flipper` (`f ← spawn (coin_flipper ()) (,c)`). With both processes

running, the function requests access to the `Coin`, in the same way as `coin_flipper`. It submits a request, reads the transmitted value, and then shares it again to release the connection. After unlocking `c` and making it available for other processes, the instruction `send d v1` sends the previously read state of `Coin` to `d` (corresponding to `int ^`). Finally, before signalling its closure (`close c`, correctly concluding the declared protocol of `d`, `@`), the function waits for `coin_flipper` to terminate by executing `wait f`.

```
;; m ← {
        c ← spawn (nd_choice ()) (,);

        v1:int ← recv c;
        print v1;

        wait c;
        close m
};;
```

Figure 3.14: Type main process for the shared coin example.

Finally, Figure 3.14 presents the main function of our program. It begins with $c \leftarrow spawn\ (nd\_choice())(,)$, where the main process spawns its "worker" and then awaits the result in the following line ($v1 : int \leftarrow recv\ c$). To observe the outcome, the value is printed, after which the main process waits for the worker to terminate ($wait\ c$) before completing its own execution ($close\ m$). The typechecker validates the interactions with the channel c ($int \wedge @$) and with m (@), confirming them as correct according to the declared types.

3.3.3 Shared types verification

With the introduction of shared types, the typechecker needs to perform one additional verification (although this feature has been implemented, it has not yet been thoroughly validated). A shared process should always be spawned as shared and return to its initial state; however, the parser allows a process to change freely between linear and shared states. Moreover, we said a shared process should always return to its initial state since it does not know how many connections are about to be opened. For these reasons, the typechecker must recheck the entire program and verify that the shared processes remain consistent throughout. It will check all the shared processes spawned during the program's execution and verify that all of them return to their initial state. The opposite condition is also verified; every shared process present at the end of the verification must have been spawned as a shared process. Along with these conditions, the program is also verified to ensure that only processes spawned as shared can return to a shared state (by invoking the `closeConnection` construct).

3.4 Compiler

The final component of the project to be adapted was the compiler. After completing the parser and typechecker, the next step is to transform programs written in our custom language into executable code. Each session type is translated into a deterministic state machine, which is then mapped onto Go's concurrency model through goroutines and channels. The compiler ultimately produces a self-contained Go program where every channel interaction and termination is guaranteed to respect the session type discipline. The resulting code is type-safe and faithfully reflects the communication protocol described in the source program. The Go program is also deadlock-free if it contains only linear processes; however, such a guarantee cannot be made for programs that include shared processes.

To better understand the working of our compiler, the following figures (Figure 3.15, Figure 3.16, Figure 3.18, and Figure 3.19) show the result of the compiler for the program described in Figure 3.10.

```
package main

import (
    "fmt"
)
```

Figure 3.15: Package declaration and necessary imports in Go.

Figure 3.15 declares the package of the program and specifies which imports are needed (`"fmt"` is the default library for printing values). This portion of code will be the same in the majority of programs; however, shared examples will require an additional library import.

```
func  main() {
   c := init_state_2(make(chan interface{}))
   go func() { c.Recv() }()
   func(c *_state_2) {
      d := init_state_2(make(chan interface{}))
      go delegate()(d)
      d.Recv()
      c.Send(nil)
   }(c)
}
```

Figure 3.16: Generated Go code for the main function.

This section adopts the strategy opposite to that of the previous chapter. To better

understand the workings of the typechecker, it was easier to start with the function declaration and move our way down (the file) to the workings of the main. Due to the complexity of the state machine, however, it is more understandable to explain how the compiler works if the explanation starts in the main function, and only then do we move upwards to the subsequent functions, outlining the state machine's states as they appear. Written in Figure 3.16, the main starts with the creation of `state_2` (`c := init_state_2(make(chan interface{}))`). `c` becomes the channel to invoke `state_2` functions.

```
type_state_2 struct {
    c chan interface{}
}

func init_state_2(c chan interface{}) *_state_2 { return &_state_2{c} }
func (x *_state_2) Send(v interface{}) { x.c <- v }
func (x *_state_2) Recv() interface{
   return <-x.c
}
```

Figure 3.17: State machine functions and state ($type_state_2$).

`state_2` as seen in Figure 3.17, is fairly simple, it is a struct that registers `c` from which it is possible to launch both the functions `(x *_state_2) Send` and `(x *_state_2) Recv`. When the function `init_state_2(c chan interface{})` is called, it receives a channel and associates it as the `state_2` struct's channel before returning its reference. The next step in Figure 3.16 is the line: `go func() { c.Recv() }()`. The main function launches a goroutine that calls the `Recv` function; the goroutine will simply be waiting to receive something through the channel `c`. The following lines of the function main declare and launch another goroutine to run concurrently. At the end of such functions, there is the interaction `c.Send(nil)` that refers to the same `state_2 c`. It will `send` a `nil` value, which is read by the first process running and waiting, and closes the channel. This is the background work the compiler lays down to ensure synchronization. Furthermore, in the custom program built, it is equivalent to the interactions of the channel `c` (`close c`) for the custom main function in Figure 3.10. The actual body of the function launched of the most simplified version of our program works similarly. Before the `c.Send(nil)` seen before, the goroutine launches another process running the `delegate` function (`go delegate()(d)`), after creating a `d` (`d := init_state_2(make(chan interface{}))`) channel, and awaits its signal in `d.Recv()`.

Moving up in Figure 3.10, the called worker, denoted as `delegate`, appears; its objective was to summon another process running `pass` and send a value (`1`) to it through the new

```
func pass() func(_x *_state_0) {
   return func(c *_state_0) {
      v1, c0 := c.Recv()
      c1 := c0.Send(v1)
      c1.Send(nil)
   }
}

func delegate() func(_x *_state_2) {
   return func(c *_state_2) {
      d := init_state_0(make(chan interface{}))
      go pass()(d)
      d0 := d.Send(1)
      _, d1 := d0.Recv()
      fmt.Printf("%v\n", 0)
      d1.Recv()
      c.Send(nil)
   }
}
```

Figure 3.18: Custom functions, pass and delegate, translated into Go.

channel created alongside the process. Then wait for a response through the same channel, print another value (`0`) and finally wait for the process running `pass` to end before closing itself. The same function rewritten in Go is clearly similar. Figure 3.18 shows `delegate` as a function that receives the pointer (`state_2`) through which is expected to communicate its end (`c.Send(nil)`). It starts its execution with the creation of a `state_0` struct (`d := init_state_0(make(chan interface{}))`) and passing it when it summons the `pass` process. The following step is to send the value 1 (`d0 := d.Send(1)`) and by doing so progresses to the next state. The `(x *_state_0) Send(v int) *_state_1` method returns a pointer to it, and as such, we use it to proceed with the execution. `_, d1 := d0.Recv()` is where the `delegate` receives, and in this case, ignores the integer from `pass` along with the pointer to the next state. Then, the value `0` is printed and finally, `delegate` awaits the end of `pass` before concluding itself.

The last function the compiler translated is the `pass` function. `pass` needs to receive a session-type in the form of `state_0` where it awaits to receive a value `v1` along with the next state of the session `c0` (`v1, c0 := c.Recv()`). `c1 := c0.Send(v1)` passes the value read in `v1` back through the new `c0`, and the communication proceeds through `c1`. Lastly, this process terminates after its last communication is accepted in `c1` (`c1.Send(nil)`).

The Figure 3.19 has the remaining states used for communication. Both `state_0` and `state_1` are structured similarly; they have a channel through which messaging is done

```
type _state_1 struct {
   c chan interface{}
   next *_state_2
}

func init_state_1(c chan interface{}) *_state_1 { return &_state_1{c, nil} }
func (x *_state_1) Send(v int) *_state_2 {
   if x.next == nil { x.next = init_state_2(x.c) }
   x.c <- v
   return x.next
}
func (x *_state_1) Recv() (int, *_state_2) {
   if x.next == nil { x.next = init_state_2(x.c) }
   return (<-x.c).(int), x.next
}

type _state_0 struct {
   c chan interface{}
   next *_state_1
}

func init_state_0(c chan interface{}) *_state_0 { return &_state_0{c, nil} }
func (x *_state_0) Send(v int) *_state_1 {
   if x.next == nil { x.next = init_state_1(x.c) }
   x.c <- v
   return x.next
}
func (x *_state_0) Recv() (int, *_state_1) {
   if x.next == nil { x.next = init_state_1(x.c) }
   return (<-x.c).(int), x.next
}
```

Figure 3.19: State machine functions and states.

and a pointer to the next state. Since both steps of the protocol involve sending a value from one process to another, they also share the same methods (`Send` and `Recv`). As the name implies, one sends a value to a channel, while the other reads that value. One thing to note is that they create their next channel only if the other end of the communication has not already created one, as they refer to the same channel in the same state.

To build a Go file like this, internally, the compiler starts by translating the program into a state graph in the form of a tree map of states. Each state is a point in the

```
package main

import (
   "fmt"
   "sync"
)
```

Figure 3.20: Package declaration and necessary imports in Go for the coin example.

communication protocol, while transitions are defined by the communication actions (like the send/receive pair). The linear operations are mapped into the `Comm`, `Choice`, and `End` categories, and a new category was added to support shared operations (`Sync`). It was possible and considered to include shared operations into the `Comm` category, but, for clarity, it is better to differentiate the operations more clearly. The synchronization naturally aligns with Go's concurrency model. For synchronized channels, when multiple goroutines or clients attempt to receive the rights to a connection, Go ensures that only one can receive it, as long as the shared process (server) only sends it once at a time. Another important condition to ensure is that, after following the entire protocol for a single execution, the channel from which the shared process reads remains unchanged; otherwise, other clients would not be able to connect.

The shared program discussed before (in Figure 3.12, Figure 3.13, and Figure 3.14) illustrates all the new operations and graph states. It starts in Figure 3.20 where a new import is added. `"sync"` will allow the use of Mutexes and synchronize multiple goroutines attempting to execute the same operation.

```
func main() {
   m := init_state_3(make(chan interface{}))
   go func() { m.Recv() }()
   func(m *_state_3) {
      c := init_state_4(make(chan interface{}))
      go nd_choice()(c)
      v1, c0 := c.Recv()
      fmt.Printf("%v\n", v1)
      c0.Recv()
      m.Send(nil)
   }(m)
}
```

Figure 3.21: Generated Go code for the `main` function.

The final result for the Coin example Figure 3.21 is very similar to the linear example. Of course, the differences do not come from this being a shared example. The function spawned in the main is linear, the main itself is a linear process, and the differences are

simply the result of a slightly more complex main written. Now, it receives a value `v1` (`v1, c0 := c.Recv()`) and prints it in the following line instead of simply waiting for the spawned process to do its work (this code can obviously be refactored into a delegate process as in Figure 3.10, however, it would bring no benefit, and instead, another complexity layer explaining it).

```
type _state_3 struct {
    c chan interface{}
}
func init_state_3(c chan interface{}) *_state_3 { return &_state_3{c} }
func (x *_state_3) Send(v interface{}) { x.c <- v }
func (x *_state_3) Recv() interface{} { return <-x.c }
```

Figure 3.22: State machine functions and state ($type_state_3$).

The program's end state is now `state_3`, Figure 3.21, instead of `state_2` as in Figure 3.16. However, only its *id* is different, comparing the late Figure 3.17 with Figure 3.22; both the state struct, methods, and usage remain the same. This is expected, as they are translations of the same operations. The different *id* is simply the result of the position in the state graph being different.

The goroutine spawned in Figure 3.21 is running `nd_choice` (Figure 3.23). `nd_choice` is a linear function that starts by spawning a `coin_head` (`go coin_head(0)(c)`) along the channel `c` (`c := init_state_0(make(chan interface{}))`). The next step is to spawn the other concurrent process, `go coin_flipper()(f, c)`, that will be competing to access `c`, along the channel `f` (`f := init_state_3(make(chan interface{}))`).

After the goroutines are launched, `nd_choice` will await permission to interact with `coin_head` through `_, c0 := c.Request()`. Then, when permission is granted, the process receives `c0` a channel in linear state from which to communicate linearly in the next stage of the protocol. In `v1, c1 := c0.Recv()`, `nd_choice` will receive a value symbolizing the state of the coin and pass it to the main function in `d0 := d.Send(v1)` later. Before communicating with the main, however, it is important to sever the connection with `coin_head` in order to free it for the `coin_flipper` to proceed with its execution (`_ = c1.Share()`).

Lastly, `nd_choice` waits for its linear process to end its execution with `f.Recv()` and communicates its own execution's end (`d0.Send(nil)`).

The final linear goroutine running along this example is `coin_flipper`. This one uses the same methods to interact with `coin_head`. It starts with a request for a private channel, `_, c0 := c.Request()`, then discards the current state of coin, semantically flipping it (`_, c1 := c0.Recv()`), and finally severs the connection with `_ = c1.Share()`.

Regarding the shared process, the goroutine running `coin_head` becomes more complex

```
func coin_flipper() func(_x *_state_3, c *_state_0) {
   return func(d *_state_3, c *_state_0) {
      _, c0 := c.Request()
      _, c1 := c0.Recv()
      _ = c1.Share()
      d.Send(nil)
   }
}

func nd_choice() func(_x *_state_4) {
   return func(d *_state_4) {
      c := init_state_0(make(chan interface{}))
      go coin_head(0)(c)
      f := init_state_3(make(chan interface{}))
      go coin_flipper()(f, c)
      _, c0 := c.Request()
      v1, c1 := c0.Recv()
      _ = c1.Share()
      d0 := d.Send(v1)
      f.Recv()
      d0.Send(nil)
   }
}
```

Figure 3.23: Custom function `nd_choice` translated to Go.

than the previous examples. It starts very similarly, to accept a request, the goroutine invokes the operation `c0 := c.OpenConnection()`. Through `c0`, the connected linear channel, it will follow the protocol by sending the integer state of coin (`c1 := c0.Send(n)`) and the next step ot takes is to wait for the client to sever the connection (`_, c2 := c1.CloseConnection()`).

These are the lines responsible for the main interaction with the clients. However, the following action is necessary to return the process to its initial state, spawn a new `coin_head`, and redirect the remaining requests to it. `c2 = c` will "restore" the channel and allow reading pending requests from clients. As done in `nd_choice`, to spawn a new `coin_head` the process creates its associated channel at the state_0 (`d := init_state_0(make(chan interface{}))`) followed by the associated goroutine in `go coin_head((n+1))(d)`.

The only thing left to do is the actual forwarding. In an infinite cycle, the current process will act as a server for the clients; at the same time, it will also serve as a client for the newly spawned `coin_head`. `c3 := c2.OpenConnection()` will open a connection

```
func coin_head(n int) func(_x *_state_0) {
   return func(c *_state_0) {
      c0 := c.OpenConnection()
      c1 := c0.Send(n)
      _, c2 := c1.CloseConnection()
      c2 = c
      d := init_state_0(make(chan interface{}))
      go coin_head((n+1))(d)
      for {
         c3 := c2.OpenConnection()
         _, d0 := d.Request()
         c3d0, d1 := d0.Recv()
         c4 := c3.Send(c3d0)
         _ = d1.Share()
         _, _ = c4.CloseConnection()
         c2 = c
      }
   }
}
```

Figure 3.24: Shared function `coin_head` translated to Go.

with the next client, while `c1 := c0.Send(n)` requests a connection to forward the client's request. `c3d0, d1 := d0.Recv()` stores the present state of the coin in `c3d0` and the following line spreads it to the client (`c4 := c3.Send(c3d0)`). Finally, the process severs its connection with its server (`_ = d1.Share()`) just before waiting to be freed too (`_, _ = c4.CloseConnection()`) and once again making a reset to read the initial channel with `c2 = c`.

Lastly, the figures below (Figure 3.25, Figure 3.26, Figure 3.27, Figure 3.28, Figure 3.29, and Figure 3.30) show the states and transitions other than the `state_3` of Figure 3.22. One important guarantee the compiler gives is that after a process's request is accepted, a client receives a channel to proceed with communication. This is achieved by creating the main `chan(lc chan interface{})` and one additional linear channel in the declaration of the struct of the state `lc chan interface{}`. Focusing on Figure 3.25, when `openConnection` returns the pointer to the next state (`state_0`), `next` is created upon `lc`, keeping `c` open for future clients and ensuring `lc` is only used in linear communications. Once the shared process is freed, it can accept future requests through `c` again.

```
type _state_2 struct {
    c             chan interface{}
    lc           chan interface{}
    next         *_state_0
    mtx           sync.Mutex
    initial_state *_state_2
}
func init_state_2(c chan interface{}) *_state_2 {
    s := &_state_2{c, make(chan interface{}), nil, sync.Mutex{}, nil}
    s.initial_state = s
    return s
}
func (x *_state_2) OpenConnection() *_state_0 {
    if x.next == nil {
        x.next = init_state_0(x.lc)
    }
    x.c <- struct{}{}
    return x.next
}
func (x *_state_2) Request() (interface{}, *_state_0) {
    if x.next == nil {
        x.next = init_state_0(x.lc)
    }
    return <-x.c, x.next
}
func (x *_state_2) Share() *_state_2 {
    x.mtx.Lock()
    defer x.mtx.Unlock()
    x.c <- struct{}{}
    return x.initial_state
}
func (x *_state_2) CloseConnection() (interface{}, *_state_0) {
    x.next = init_state_0(x.c)
    return <-x.c, x.next
}
```

Figure 3.25: Generated state machine definitions in Go for state_2 (part 1)

```
        func (x *_state_2) Send(v interface{}) *_state_0 {
            if x.next == nil {
                x.next = init_state_0(x.c)
            }
            x.c <- v
            return x.next
        }
        func (x *_state_2) Recv() (interface{}, *_state_0) {
            if x.next == nil {
                x.next = init_state_0(x.c)
            }
            return <-x.c, x.next
        }
```

Figure 3.26: Generated state machine definitions in Go for state_2 (part 2)

```
type _state_1 struct {
    c    chan interface{}
    next   *_state_2
}
func init_state_1(c chan interface{}) *_state_1 { return &_state_1{c, nil} }
func (x *_state_1) Send(v int) *_state_2 {
    if x.next == nil {
        x.next = init_state_2(x.c)
    }
    x.c <- v
    return x.next
}
func (x *_state_1) Recv() (int, *_state_2) {
    if x.next == nil {
        x.next = init_state_2(x.c)
    }
    return (<-x.c).(int), x.next
}
```

Figure 3.27: Generated state machine definitions in Go for state_1.

```
type _state_0 struct {
    c    chan interface{}
    lc   chan interface{}
    next   *_state_1
    mtx   sync.Mutex
    initial_state *_state_0
}
func init_state_0(c chan interface{}) *_state_0 {
    s := &_state_0{c, make(chan interface{}), nil, sync.Mutex{}, nil}
    s.initial_state = s
    return s
}
func (x *_state_0) OpenConnection() *_state_1 {
    x.mtx.Lock()
    if x.next == nil {
        x.next = init_state_1(x.lc)
    }
    x.mtx.Unlock()
    x.c <- struct{}{}
    return x.next
}
func (x *_state_0) Request() (interface{}, *_state_1) {
    x.mtx.Lock()
    if x.next == nil {
        x.next = init_state_1(x.lc)
    }
    x.mtx.Unlock()
    return <-x.c, x.next
}
func (x *_state_0) Share() *_state_0 {
    x.mtx.Lock()
    defer x.mtx.Unlock()
    x.next = nil
    x.c <- struct{}{}
    return x.initial_state
}
```

Figure 3.28: Generated state machine definitions in Go for state_0 (part 1)

```
func (x *_state_0) CloseConnection() (interface{}, *_state_1) {
    if x.next == nil {
        x.next = init_state_1(x.c)
    }
    return <-x.c, x.next
}
func (x *_state_0) Send(v interface{}) *_state_1 {
    if x.next == nil {
        x.next = init_state_1(x.c)
    }
    x.c <- v
    return x.next
}
func (x *_state_0) Recv() (interface{}, *_state_1) {
    if x.next == nil {
        x.next = init_state_1(x.c)
    }
    return <-x.c, x.next
}
```

Figure 3.29: Generated state machine definitions in Go for state_0 (part 2)

```
type _state_4 struct {
    c chan interface{}
    next *_state_3
}
func init_state_4(c chan interface{}) *_state_4 { return &_state_4{c, nil} }
func (x *_state_4) Send(v int) *_state_3 {
    if x.next == nil {
        x.next = init_state_3(x.c)
    }
    x.c <- v
    return x.next
}
func (x *_state_4) Recv() (int, *_state_3) {
    if x.next == nil {
        x.next = init_state_3(x.c)
    }
    return (<-x.c).(int), x.next
}
```

Figure 3.30: Generated state machine definitions in Go. (state_4)

4 Evaluation

Before concluding this work, it is essential to conduct an objective evaluation. The main objective of this project was to extend the custom programming language. This objective was successfully achieved, as the compiler can now write Go programs for our language with the shared types extension.

The programs written are always type-safe and faithfully reflect the communication protocol described in the source program. If such a program is entirely linear, it is deadlock-free; however, such a guarantee cannot be made for programs with shared sessions.

Performance cannot be evaluated for linear processes. Comparing the linear programs generated now with the ones before the language extension, they are the same. The changes made to linear types were done solely for the sake of consistency and to encapsulate the new types required. As such, it is not that timings cannot be measured; it simply means the performance for the same code is the same.

One interesting observation can be made, however, by writing the same program both linearly and including shared types, we can measure the impact of using this type of synchronization.

The first example is a simple program where the primary process passes an integer value (`1`) and a secondary process returns another integer (`n+1`). The print at the end will show how many times the value is passed. The structure needs to be this one in order to work exactly like the shared program.

The linear program's main (Figure 4.1) will send a value, receive another one that will be printed, and make a final synchronization to ensure the linear process that was spawned has terminated.

Finally, after using the compiler to translate our program into Go, I made a few changes to test the program's performance more effectively. Refactoring the main function of the program, it is run 100 times consecutively. For each execution, the time is measured and stored. Ultimately, I analyze the average time (*mean*) of each execution, the median value, after ordering the execution times, and select the one in the middle. Finally, I also present

```
stype PlusOneType int => int ^ @;

plusOne : unit -> {PlusOneType}
fun u -> c <- {
    n:int <- recv c;
    send c (n+1);
    close c
}
end;

;;m <- {
    d <- spawn (plusOne ()) (,);
    send d 1;

    v1:int <- recv d;

    print v1;
    wait d;
    close m
};;
```

Figure 4.1: Plus one example (linear).

the maximum and minimum times the execution took.

Table 4.1: Execution time statistics over 100 runs for linear and shared programs (Figure 4.1 and Figure 4.2)

Statistic	Figure 4.1 (μs)	Figure 4.2 (μs)
Mean	6.112	6.888
Median (p50)	4.438	4.896
Minimum	1.750	1.708
Maximum	43.250	60.792

Looking at Table 4.1, we can see that there is a significant difference between the *mean* and *median* values. When measuring such small intervals, it is expected that some executions will have very different values. The first execution takes a significantly longer time to complete due to the task management of the Operating System or other external factors. Taking these values into account makes the *mean* slightly higher than most of the results read.

From here on out, when comparing executions for the different examples, I will refer to the *median* execution time of each program (instead of *mean*) since most of the executions are closer to it than to the *average*.

For the program Figure 4.1, the biggest execution was the first execution, as in every other example, it took 43.250μ while the fastest execution took 1.750μ; however, most of the values were near the *median* of 4.438μ.

```
sh stype PlusOneType rec x. rq int => int ^ sh (sh)x;

plusOne : unit -> {PlusOneType}
fun u -> c <- {
    openConnection c;
    n:int <- recv c;
    send c (n+1);
    closeConnection c;
    d <- sspawn (plusOne ()) (,);
    sfwd c d
}
end;

;;m <- {
    d <- sspawn (plusOne ()) (,);
    request d ;

    send d 1;
    v1:int <- recv d;
    print v1;

    share d;
    close m
};;
```

Figure 4.2: Plus one example (shared)

The shared version of this program (Figure 4.2) will, instead of ending the process, have a different task. Besides synchronization at the beginning and end of execution, the shared process will attempt to spawn a new instance to prepare for future runs. However, the main goroutine running may close after the `share`, not waiting for the spawn to occur.

In truth, the shared program only makes one extra communication. Both the `close c` (Figure 4.1) and `closeConnection c` (Figure 4.2) are translated as sending and receiving a value through a channel. Only the `openConnection c` represents an extra interaction.

Measuring the execution time of the Go program compiled from the code in Figure 4.2 one hundred times, the result obtained is similar to the previous one, taking around 4.896μ for a single execution.

Such a slight difference is more likely due to the environment's fault than to the program itself, but the following examples will further substantiate this fact.

The next performance test was done with a slight difference from Figure 4.1 and Figure 4.2. Both a linear version and a shared version were created, where I have altered the previous programs to both send and receive 100 (`n+1`) values.

Looking at Table 4.2, the difference between executing a larger linear program inside another linear process or a shared process instead is negligible. Even if 51.146μ is smaller than 53.146μ, despite the difference being small, both the *minimum* and *maximum* values

Table 4.2: Execution time statistics over 100 runs for linear and shared programs with 100 send/receive operations

Statistic	linear program (μs)	shared program (μs)
Mean	57.34	58.022
Median (p50)	51.146	53.146
Minimum	36.542	34.042
Maximum	191.542	150

are smaller in the shared program. Nevertheless, that is also within expectations. As long as there is no concurrent access to the same shared process, there will be no noticeable difference in execution times.

As one last comparison, instead of executing one hundred `send` and `receive` operations, the following program will have only ten (total operations) but will be called multiple times. The linear example will spawn multiple processes one after the other, while the shared example will be acquired and freed multiple times.

Table 4.3: Execution time statistics over 100 runs for linear and shared programs ten spawns/requests each

Statistic	linear program (μs)	shared program (μs)
Mean	21.223	74.475
Median (p50)	19.062	74.475
Minimum	12.166	39.917
Maximum	48.792	128.875

The results in Table 4.3 were surprising at first. The shared example may seem to hold an advantage at first because the spawn of a new process to begin the next cycle can occur as long as the main executes its `share` and concurrently with the main. In this case, the main does not have another execution after freeing a connection to request a new one, so this advantage is not applied. The print could be called in between, but there are two reasons it is not. The first one is that printing a value is a lightweight operation and does not significantly change the result. The second reason is that it could be a possible improvement over the linear version; however, at the same time, it would make the examples differ from one another and possibly skew the results instead of facilitating a simple comparison in performance.

Regarding the reason for this occurrence, suspicions were identified by adding more prints (Figure 4.3).

Understanding how to sever a shared connection and how it becomes ready for the next cycle. It is understandable to have a linear incremental value for the execution time of the same process. When a shared process returns to its initial state, it spawns a new process to execute the next cycle and forwards future requests to it.

This means that in the first execution, it is done directly between client and server,

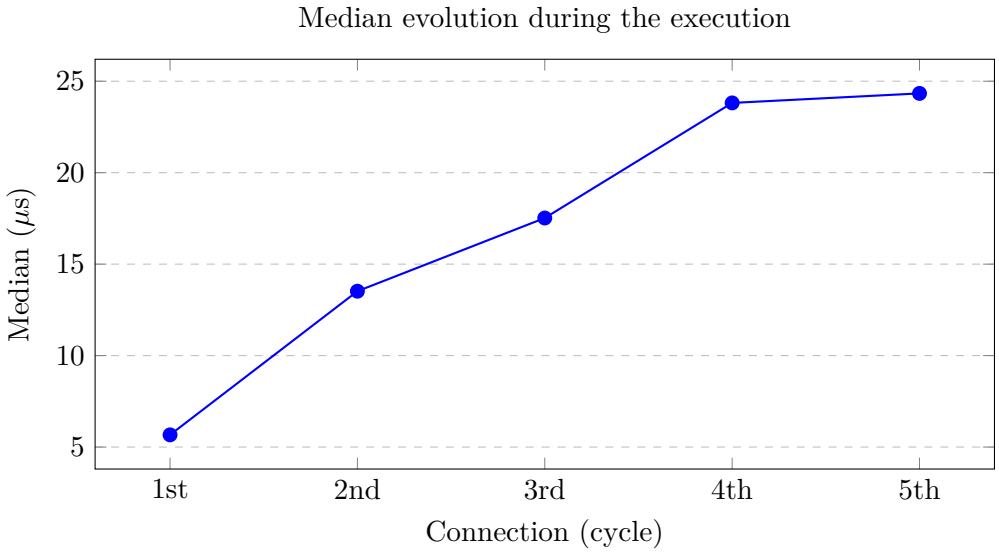

Figure 4.3: Median evolution during the execution of the same cycle multiple times

the next one has an intermediate process forwarding the operations, the third already has two middle agents, and so on (leading to the result shared in Figure 4.3).

Although this forward pass seems simple, when the compiler generates the *Go* code, each middle agent will simultaneously act as a server to its client and as a client to its server, thereby doubling the operations that the original client and original server perform.

This way of execution is more faithful to the language, but, as a future improvement, maybe a new operation should be added or an optimization when writing `sfwd`'s *Go* code.

Having discovered that the number of executions of a single shared process has a significant impact on execution time, the last thing we need to evaluate is the influence of the number of processes competing for ownership of a single shared channel.

Still using the first example (Figure 4.2) as a starting point, the main process will now be responsible for spawning workers. One of these workers will be equivalent to `plusOne` (with ten linear operations instead of two) in the shared process. In contrast, the others will execute the corresponding linear operators to access this goroutine. The variable in this test will be the number of concurrent processes trying to access the server.

In Figure 4.4, we can see the beginning of an exponential curve, despite requesting multiple accesses to the same shared process incrementing the time cost linearly. When we add the linear cost for multiple accesses to those conditions, it comes with a significant cost.

In conclusion, adopting a linear approach whenever possible is recommended; however, shared types bring a whole new world of possibilities. This increment in the language expression does not come solely with the dangers of deadlocks that the compiler can no longer guarantee do not happen, but also with an incremental cost in execution time.

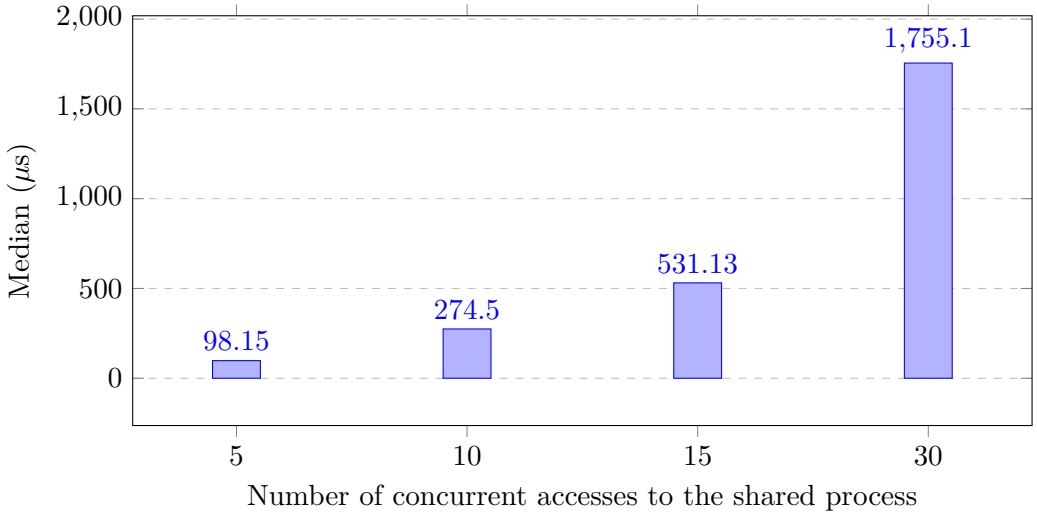

Figure 4.4: Median for execution times varying the number of concurrent acesses to the same shared resource

5 Conclusion

In this study, based on the work of [3], an extension to the language previously developed by [2] was implemented. This implementation involves adding new functionality that allows a process to spawn another, and multiple processes can request access to the same connection. There is a cost associated with the freedom of using shared types, however. Our compiler will no longer guarantee a deadlock-free result when compiling programs that use shared types.

This extension to the language was made by creating new types and process expressions. The typechecker now supports the new operations and ensures that the final result is type-safe and follows the written protocol correctly.

The changes to the compiler successfully generated a Go program supporting shared processes. Similar to linear processes, they run on a dedicated goroutine; however, different clients can access them, one at a time. It also ensures the creation of a dedicated linear connection from which accepted clients can proceed with their execution. At the same time, once the connection is severed, it is guaranteed to return to the initial state and await other connections from the original shared channel.

Future work can focus on reducing the risk of deadlocks or signaling when a program is unsafe. For example, a program with a single shared process will always be deadlock-free as long as the protocol is followed correctly (which the typechecker already guarantees). On the other hand, there is no restriction on the order in which shared processes are requested, which is the leading cause of deadlocks. Establishing a hierarchy between shared processes would mitigate this problem. Another approach could be to restrict the use of shared processes to `unsafe` code blocks, similarly to what happens in *Rust*. This way, the programmer would be forced to assume the risk of using shared processes by marking where such risks are allowed.

Another improvement that can be made to this work is to improve its performance. As seen in chapter 4, shared processes can be time-consuming when multiple executions of the same process are needed. In linear examples, `fwd` is not an operation expected to happen

too many times, so its lack of performance can be ignored. However, for shared processes, `sfwd` is the only way to complete one "cycle," and since it is designed to run endlessly, it should be optimized. One way to achieve this would be to switch from a functional to an imperative approach. Instead of endlessly spawning a new process, it could be translated to an infinite cycle when translated to *Go*.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[2] J. a. Geraldo. "Making Session Types Go". In: *INFORUM* (2022), p. 55 (cit. on pp. 2, 13, 59).

[3] S. Balzer and F. Pfenning. "Manifest Sharing with Session Types". In: *Proceedings of the ACM on Programming Languages* 1 (ICFP 2017-08-29), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3110281. (Visited on 2023-01-02) (cit. on pp. 2, 5, 15, 16, 19, 23, 24, 28–30, 59).

[4] L. Caires, F. Pfenning, and B. Toninho. "Linear Logic Propositions as Session Types". In: *Mathematical Structures in Computer Science* 26.3 (2016-03), pp. 367–423. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129514000218. (Visited on 2023-01-02) (cit. on pp. 2, 5, 11, 12).

[5] T. Ehrhard. "An Introduction to Differential Linear Logic: Proof-Nets, Models and Antiderivatives". In: *Mathematical Structures in Computer Science* 28.7 (2018-08), pp. 995–1060. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129516000372. (Visited on 2023-02-03) (cit. on pp. 2, 17).

[6] P. Rocha and L. Caires. "Propositions-as-Types and Shared State". In: *Proceedings of the ACM on Programming Languages* 5 (ICFP 2021-08-22), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3473584. (Visited on 2023-01-03) (cit. on pp. 2, 17).

[7] GoncaloLourenco. *Sessint-Compiler*. URL: https://github.com/gmlourenco/sessint-compiler (visited on 2025-09-29) (cit. on p. 2).

[8] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems*. Ed. by C. Hankin. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 1381. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-64302-9 978-3-540-69722-0. DOI: 10.1007/BFb0053567. (Visited on 2023-01-14) (cit. on pp. 5, 7, 11).

[9] K. Honda, N. Yoshida, and M. Carbone. "Multiparty Asynchronous Session Types". In: (2008-01-07/2008-01-12) (cit. on pp. 5, 11).

[10] M. Dezani-Ciancaglini et al. “Session Types for Object-Oriented Languages”. In: *20th European Conference on Object-Oriented Programming (ECOOP)*. Vol. 4067. Lecture Notes in Computer Science. Springer, 2006, pp. 328–352. DOI: 10.1007/11785477_22 (cit. on p. 5).

[11] R. Hu, N. Yoshida, and K. Honda. “Session-Based Distributed Programming in Java”. In: *22nd European Conference on Object-Oriented Programming (ECOOP)*. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 516–541. DOI: 10.1007/978-3-540-70592-5_22 (cit. on p. 5).

[12] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. “Session Types for Rust”. In: *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*. ACM, 2015, pp. 13–22. DOI: 10.1145/2808098.2808101 (cit. on p. 5).

[13] R. Neykova and N. Yoshida. “Multiparty Session Actors”. In: *16th International Conference on Coordination Models and Languages (COORDINATION)*. Vol. 8459. Lecture Notes in Computer Science. Springer, 2014, pp. 131–146. DOI: 10.1007/978-3-662-43376-8_9 (cit. on p. 5).

[14] A. Scalas and N. Yoshida. “Lightweight Session Programming in Scala”. In: *30th European Conference on Object-Oriented Programming (ECOOP)*. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 21:1–21:28. DOI: 10.4230/LIPIcs.ECOOP.2016.21 (cit. on p. 5).

[15] L. Caires and F. Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *CONCUR 2010 - Concurrency Theory*. Ed. by P. Gastin and F. Laroussinie. Red. by D. Hutchison et al. Vol. 6269. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 222–236. ISBN: 978-3-642-15374-7 978-3-642-15375-4. DOI: 10.1007/978-3-642-15375-4_16. (Visited on 2023-01-29) (cit. on pp. 5, 11, 14, 17).

[16] B. Toninho. “A Logical Foundation for Session-based Concurrent Computation”. In: () (cit. on p. 5).

[17] P. Wadler. *Propositions as Sessions*. 2012 (cit. on pp. 5, 17).

[18] B. Toninho, L. Caires, and F. Pfenning. “Higher-Order Processes, Functions, and Sessions: A Monadic Integration”. In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Red. by D. Hutchison et al. Vol. 7792. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 350–369. ISBN: 978-3-642-37035-9 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_20. (Visited on 2023-01-02) (cit. on pp. 5, 6).

[19] N. Kobayashi. “A Partially Deadlock-freemped Process Calculus”. In: () (cit. on p. 11).

[20] W. Kokke and O. Dardha. *Prioritise the Best Variation*. 2022-12-23. arXiv: 2103.14466 [cs]. URL: http://arxiv.org/abs/2103.14466 (visited on 2023-01-02). preprint (cit. on p. 17).

[21] S. Lindley and J. G. Morris. "A Semantics for Propositions as Sessions". In: *Programming Languages and Systems*. Ed. by J. Vitek. Vol. 9032. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 560–584. ISBN: 978-3-662-46668-1 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_23. (Visited on 2023-02-06) (cit. on p. 17).

[22] P. Wadler. *Propositions as Types*. 2014 (cit. on p. 17).