DEPARTMENT OF
COMPUTER SCIENCE

DAVID FILIPE RODRIGUES MIRA

BSc in Computer Science

A COMPILER FOR CLASSICAL SESSION TYPES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2000

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

A COMPILER FOR CLASSICAL SESSION TYPES

DAVID FILIPE RODRIGUES MIRA

BSc in Computer Science

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

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2000

A Compiler for Classical Session Types

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.5) [20].

To those broken, yet unbowed

Acknowledgements

When embarking on this journey I never could've imagined the life lessons and values I would learn from it. It was a long and arduous journey, but I've come out on the other side a better person. As such I would like to those who not only helped, but made it possible. I was never meant to be a wordsmith, so I'll keep this short.

To start I would like to properly, and beyond my capability, express my sincere thanks to my advisor, Professor Bernardo Toninho, who endured me for countless months. Tirelessly and always understanding, he single-handedly made this thesis possible. Despite my many failings, technical or otherwise, during the execution of this work he would always help me find a solution to the problems ahead, and even went beyond his responsibilities to ensure that I was more than well accompanied on this campaign. Thank you beyond measure, and I hope all the best to you and yours.

Second I would like to thank my friends, without whom I would've probably either never finished this work, or finished it sooner. Nonetheless I thank them, for always being by my side, and making me a better person. In particular I would like to thank my favorite biologist, Julio, who is also finishing a thesis and so understands better than the rest what it means. I would also like to thank him for the countless philosophical discussions we had over coffee, which probably, did more harm than good on the overall completion of this work.

I leave thusly my dearest mother for the end. Without question one of the most important people in my life. Not only did she make that a possibility, she did more by always being there for me, through the highs and lows that both this journey and life itself bring. She always listens to what I have to say without judgement, be it just my frustrations, crazy ideas, or resentments. I don't think it's possible for me to disappoint her, and believe me I have tried. There are countless more things I could say about my mom, but I did promise to keep this short.

I would also like to thank my dad, although we do have our disagreements, I would not be who I am without you. For that I thank you. For the education you gave me, and the countless fond memories. I would also like to thank you for always providing for me and being more than a father, a friend, regardless of everything.

”

“Why do we fall?
So we can learn to pick ourselves up.”

— Alfred Pennyworth, Batman Begins
(Butler)

Abstract

The ever-increasing need to communicate information between different systems is an obvious and quite necessary consequence of the exponential growth of offered services and the end-user population. This has led to an increase in concurrency based systems being used in the industry.

Unfortunately the concurrent programming paradigm is error prone due to the huge amount of variability, inherent to this style of programming originating from the use of shared resources, or unordered execution of tasks along the many processing units. As a consequence of this variability programs can easily deadlock, produce incorrect results or incur incorrect behaviour. There are many solutions to surmount these difficulties and ensure correct behaviour, message-passing being one of the most used, and the most relevant for this work.

For the purposes of this work we are interested in the message-passing solution, mainly because these can be implemented using Session Types, a typing discipline that guarantees compile-time correctness by forcing the communication channels to adhere to a specified protocol. This paradigm can be further improved with the introduction of *Linear Logic* (in our case the Classical interpretation of Linear Logic), thus ensuring us of the absence of deadlocks in our programs.

As a part of our solution to this problem, we intend to explore the tight relationship between Classical Linear Logic (CLL) and Session Types by implementing a natively "logically" session-typed language along with its associated type checker, and a compiler based on the one implemented by J. Geraldo [14], capable of compiling our language into Go. We also aim to extend this language with some constructs such as the Shared Mutable State and make a comparative study between the two interpretations of Linear Logic (Classical and Intuitionistic) in order to better understand the advantages and disadvantages of each implementation.

Keywords: Linear Logic, Concurrency, Compilation, Session types

Resumo

O constante aumento da necessidade de comunicar informação entre diferentes sistemas é uma consequência óbvia e bastante necessária do aumento exponencial de serviços oferecidos e utilizadores finais. Isto tem resultado num aumento da utilização de sistemas baseados em concorrência na indústria.

Infelizmente o paradigma da programação concorrente é erróneo devido à inerente variabilidade deste estilo de programação, resultante do uso de recursos partilhados ou execução desordenada de tarefas ao longo de todas as unidades de processamento. Como consequência destas características, os programas concorrentes podem fácilmente resultar em *deadlocks*, incorrer em comportamento incorreto ou devolver resultados que não fazem sentido. Existem muitas formas de circundar estes problemas e assegurar um comportamento correto, sendo que uma das mais utilizadas, e a mais relevante para este trabalho, são os sistemas de *message-passing*.

Os sistemas de *message-passing* podem ser implementados utilizando *tipos de sessão*, uma disciplina de tipagem que nos permite defenir um protocolo ao qual um canal de comunicação terá de obedecer, como consequência este sistema garante-nos a correctude dos programas durante a compilação. Conseguimos ainda aumentar as garantias deste sistema através da introdução da *Lógica Linear* (no nosso caso Lógica Linear Clássica), garantindo-nos assim também a ausência de *deadlocks* nos nossos programas.

Como parte da solução apresentada por este trabalho, tencionamos explorar a ligação entre os Tipos de Sessão e a Lógica Linear Clássica, implementando para isso uma linguagem que nativamente utiliza Tipos de Sessão "lógicos", juntamente com o *type checker* associado e um compilador baseado na implementação feita por J. Geraldo [14], capaz de compilar a nossa linguagem para Go. Outros objetivos do nosso trabalho passam por extender a linguagem com alguns conceitos como o Estado Mutável Partilhado e a produção de um estudo comparativo entre as duas intrepretações da Lógica Linear (Clássica e Intuicionista) para melhor entender as vantagens e desvantagens das duas implementações.

Palavras-chave: Lógica Linear, Concurrencia, Compilação, Tipos de sessão

Contents

List of Figures

List of Listings

Acronyms

ASTs Abstract Syntax Trees *(p. 82)*

CLASS *(pp. ix, 3, 17, 18, 21–23, 27)*

CLL Classical Linear Logic *(pp. v, 7, 13, 31, 33, 44, 46, 54, 55)*

DiLL Diferential Linear Logic *(p. 21)*

ILL Intuitionistic Linear Logic *(pp. 31, 45, 54)*

LST Logical Session Types *(pp. 13, 14)*

1 Introduction

1.1 Motivation

Throughout history the need for more processing power has been answered by the development of better and more powerful processing units. This has resulted in an increase in multi-core architectures over the last two decades. As such programs also needed to adapt to make use of the superior computational power and an obvious solution that would allow programs to take full advantage of these architectures, is the concurrent programming paradigm.

Concurrent programming is based on the ability to share information and computation power between systems in order to process bigger or more complex tasks in a shorter amount of time. This is usually done by dividing a process into multiple threads, resulting in each thread working only on a smaller portion of the workload and thus greatly increasing the overall efficiency of the process. Unfortunately, this technique may complicate the task itself by introducing opportunities for problems like deadlocks, data races and incorrect orders of operations to occur which can lead to the program terminating in erroneous results or not terminating at all.

To solve these issues some solutions have been created, shared memory and message-passing are two of the most commonly used paradigms in concurrent programming to deal with such issues. Shared memory relies on all of the components that are executing some task concurrently to share the same memory space, and accessing it with some conditions in order to prevent data racing conditions. Communication between multiple systems can, and often is also executed in this way to improve the efficiency of communication processes.

Another way to solve the issues that comes with the territory of the concurrent programming paradigm is message-passing. This technique involves forcing the parallel components in question to exchange information, data or instructions through messages, which are units of information containing the relevant details for operation.

Message-passing is usually done using channels that connect two processes, where one process sends a message and the other one receives it, this can be done synchronously

or asynchronously. Message-passing is the paradigm that most interests us in this work because we can couple it with *Session Types*, a typing discipline that allows the specification of a protocol to which a certain communication channel must adhere to. This technique of "channel typing" prevents typing errors from occurring when exchanging messages and data between processes, by ensuring that all the data exchanged through a channel is in the correct order and of the correct type, thus ensuring the correctness of our programs at compile-time. Alongside this assurance of strong safety, a session-typed language usually also ensures session fidelity and confluence.

Although Session Typed languages have many advantages, they do not solve all the problems and complexities that come with concurrency. It is still possible for a program to deadlock even under all the scrutiny of a session typed language, due to the fact that multiple processes can share multiple channels between them. This is where *Linear Logic* can be useful. Because it treats its variables as resources that must be used *exactly* once, if we integrate Linear Logic into the implementation of our Session Types then we can ensure that every channel is used exactly once, in other words, although processes can have multiple channels associated to them, no two processes will share more than one channel between them, making it impossible for deadlocks to occur.

Although the *Intuitionistic* interpretation of Linear Logic is generally more popular when it comes to implementing a "linearly-typed" language, in our work we will explore the connection between the *Classical* interpretation of Linear Logic and Session Types, expanding on the contributions of L. Caires, F. Pfenning and B. Toninho [9], and P. Wadler in [25], partly because this "Classical" nature allows us to include in our language other very useful constructs, like Shared Mutable State as studied by P. Rocha and L. Caires in [22].

Some other very important works involving the development and study of Linear Types and this paradigm of "Propositions as Programs" are the works of Abramsky [1, 2] and G. Bellin and P.J. Scott [6], that establish the theoretical foundations of linear types and translate them into the π-calculus. The work of L. Caires, F. Pfenning and B. Toninho [9] which explores the implementation of Logical Session Types. And finally, the work of J. Geraldo [14] where he creates the compiler on which we will base our implementation.

The products of our work would then be to:

1. A natively session-typed language, with basis in Classical Linear Logic to guarantee deadlock-freedom.

2. A type-checker to enforce the validity of programs written in our language.

3. Extend or adapt the compiler created by J. Geraldo [14] to be able to compile our language into Go.

4. Further implement some extensions based on the Classical interpretation of Linear Logic, such as Shared Mutable State as described in [22].

5. Produce a comparative study between the two interpretations of Linear Logic.

1.2 Document Organization

In Chapter 2 we contextualize our work and its foundations, by gradually introducing the concepts of Linear Logic and its two interpretations (Section 2.1) and Session Types (Section 2.2). In Section 2.3 we explain the integration of these two systems into Logical Session Types by at first introducing a "striped down" version of the CLASS language, and later (in Section 2.4.2) introducing the rest of the constructs of the language.

In Chapter 3 we introduce and describe the structure, rules, typing mechanisms as well as explain the overall design of the session-typed functional language we developed. We also make periodic comparisons to the language's inspiration and basis for our work, the language developed by J. Geraldo in [14]. At the end of this chapter we provide some examples, of varying complexity, of programs written in our language.

We then explain in Chapter 4 the implementation of the typechecker and compiler that were constructed along with the language, focusing mainly on the mechanisms introduced by us in this work, instead of a detailed explanation. This is because out implementation, like our language, is based on the already well-documented works of J. Geraldo ([14]).

We then make a brief and simple analysis, in Chapter 5, of the performance of our compiler and compare the results with previous works. Finally we present our conclusions in Chapter 6.

2

Background

The concurrent programming paradigm has intrinsic difficulties that most modern programming languages were not designed around, making it much harder to reason about results and ensure the correctness of programs than with simple sequential programming. Nevertheless the advantages of concurrent programming are clear and the demands for such a paradigm are ever increasing, and so languages have been adapting over time in order to facilitate the implementation of solutions that utilize concurrency. But this adaptation introduces many problems and inefficiencies, and so researching ways to build mechanisms to better deal with concurrency, and that are designed around such a paradigm from the get go are crucial in order to advance our understanding of the power of concurrency, and its applications in real world scenarios. We will now discuss in further detail some of these mechanisms.

2.1 Linear Logic

Linear Logic is a substructural logic derived from Classical Logic by J.-Y.Girard [15]. Being a substructural logic means that some of the basic "building blocks" of the logical proof process are not used, in the case of Linear Logic these are the *weakening* and *contraction* rules, that although not completely absent from the its lexicon, are restricted in its uses. This means that when considering the world under Linear Logic propositions can be interpreted as (resources) that can neither be discarded without concern, nor duplicated arbitrarily, and must instead be used exactly once. Another one of the key concepts of Linear Logic is it's (duality), a concept inspired by the Classical approach to logical systems, that says that every connective, has a counterpart, a *dual*, that behaves as it's opposite. And seeing that the Curry-Howard correspondence establishes that mathematical proofs can be seen as programs, this duality can be viewed as a process that receives a value, and another, it's dual, that sends it. These inherent properties of Linear Logic made it a fantastic tool for modelling systems where the usage of resources, concurrency, or state transformations play a central part, such as programming languages. An excellent example of Linear Logic being applied for its advantages is the typing discipline of Linear

Types, that utilizes the foundations and rules of Linear Logic to create a Type System that can then be used to verify the validity of concurrent programs.

Because of this "new-found" resource-awareness some of the classical logical connectives must be adapted or reinterpreted, and some new ones can be introduced. In this next section we go over all of the connectives of Classical Linear Logic and explain them briefly, as well as present the new rules associated to these connectives as defined in [5], for a more detailed explanation of these connectives refer to [15, 5, 7].

- **Additive Connectives:**
  - Disjunction ($\oplus$): Describes a connective that can provide either resource A or B, the choice of which is unknown *a priori*. For example, a machine that randomly choses a prize from a prize pool of A and B, can be modelled by A $\oplus$ B, where we will get either A or B, but we don't know which beforehand.

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \quad (\oplus_1) \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B} \quad (\oplus_2)$$

  - Conjunction (&): Describes a connective that makes two resources available, and allows us to choose which to use. For example if a merchant has two products available for purchase, then the merchant can be modelled as A & B, where a client will choose which to buy. It is important to note that the client cannot buy both of the resources in a single operation.

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B} \quad (\&)$$

- **Multiplicative Connectives:**
  - Conjunction ($\otimes$): Describes a connective that provides A and B. For example, if we have a merchant that sells 2 products, both for 1$, then the connective that describes this sale would be A $\otimes$ B.

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \quad (\otimes)$$

  - Disjunction ($\parr$): Describes a connective that requires one of A or B, but we won't know which one *a priori*, so both must be provided. For example, you are on vacation and need to power a computer, you know the vacation spot will either have outlets of type A or type B, but you won't know which one until you arrive, so you must pack both adapters.

$$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B} \quad (\parr)$$

- **Exponential Connectives:** Here is where an exception is made, and controlled access to the structural rules for weakening and contraction is given through the following structural rules:

$$\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \quad \text{(Contraction)} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A} \quad \text{(Weakening)}$$

  - Of Course (!): Describes a connective that can supply an arbitrary amount of A's, this is where weakening is reintroduced. For example if A means water, !A could mean a reusable source of water, like a tap or a well.

$$\frac{\vdash ?A_1, ..., ?A_n, B}{\vdash ?A_1, ..., ?A_n, !B} \quad or \quad \frac{\vdash ?\Gamma, B}{\vdash ?\Gamma, !B} \tag{!}$$

  - Why Not (?): Describes a connective that requests an arbitrary amount of A's, this is where contraction is reintroduced. For example if A models a food donation, ?A could mean a food bank, where we can make a potentially infinite amount of food donations.

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} \tag{?}$$

- **Linear Implication:** Instead of the classical meaning of implication, linear implication ($\multimap$) describes a resource transformation. For example, if A is an amount of money and B a product, A $\multimap$ B can model a vending machine that accepts/consumes A and returns/gives B.

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta} \quad (\multimap -Intro) \qquad \frac{\Gamma \vdash A \multimap B, \Delta \quad \Gamma' \vdash A, \Delta'}{\Gamma, \Gamma' \vdash B, \Delta, \Delta'} \quad (\multimap -Elim)$$

- **Linear Negation:** While classical negation can usually be interpreted as an absence of some literal, linear negation ($\overline{A}$) can be interpreted as a request for a resource. For example, if A is selling a product, then $\overline{A}$ can be interpreted as a need for that product. It is also important to note that like classical logical negation this negation is involutive, meaning that $\overline{(\overline{A})} = A$ for all propositions.

- **Identity Connectives:** These connectives are derived from the identities of classical disjunction (True) and conjunction (False), since a truth value has no meaning in Linear Logic, in which the point is to manage resources of a system.

  - **Multiplicatives:**

    * "True" (1): Represents the identity of the multiplicative conjunction ($\otimes$), and can be interpreted as the absence of any resource.

$$\frac{}{\vdash 1} \tag{1}$$

    * "False" ($\perp$) Represents the identity of the multiplicative disjunction ($\parr$), can be interpreted as a contradictory state, the request for two resources that can never coexist.

$$\frac{\vdash \Gamma}{\vdash \Gamma, \bot} \quad (\bot)$$

– **Additives:**

* "True" ($\top$) Represents the identity of the additive conjunction (&), and can be interpreted as a container for unneeded or unused resources.

$$\frac{}{\vdash \Gamma, \top} \quad (\top)$$

* "False" (0) Represents the identity of the additive disjunction ($\oplus$), can be interpreted as a "logical dead end", a resource that is unobtainable or impossible to produce.

$$(There\ is\ no\ rule\ for\ this\ connective)$$

Other important structural rules of Classical Linear Logic (CLL) are described next. It is important to highlight the *Cut* rule that allows us to model interaction between two processes through the *cut-elimination* theorem.

- **Exchange Rule:** Formalizes the fact that we do not care about the order in which propositions are arranged inside a context.

$$\frac{\vdash \Gamma, A_1, A_2, \Delta}{\vdash \Gamma, A_2, A_1, \Delta} \quad (\text{Ex})$$

- **Axiom:** This rule serves as a starting point for the sequent calculus.

$$\frac{}{\vdash \overline{A}, A} \quad (\text{Ax})$$

- **Cut Rule:** This rule is a way to compose proofs. In Linear Logic this rule can be viewed as making use of the cut resources.

$$\frac{\vdash \Gamma, A \quad \vdash \overline{A}, \Delta}{\vdash \Gamma, \Delta} \quad (\text{Cut})$$

As mentioned before duality plays an important role in Classical Linear Logic, instead of the *introduction* and *elimination* rules of propositional logic, we have duality, in which a connective can be seen as the introduction rule and its dual as the elimination rule for some proposition [7], and as such we get the one-sided version of the inference rules presented before. This clean balance between connectives is comparable to the sender/receiver paradigm of a parallel message-passing system, making Linear Logic a very good modelling candidate for such systems. It is also worth noting that comparison can be made between duality and negation in the sense that the dual of a proposition is

Table 2.1: Connective propositions and respective duals

Proposition (A)	Dual ($\overline{A}$)
$\overline{(A)} = \overline{A}$	$\overline{(\overline{A})} = A$
$\overline{(A \otimes B)} = \overline{A} \parr \overline{B}$	$\overline{(A \parr B)} = \overline{A} \otimes \overline{B}$
$\overline{(A \oplus B)} = \overline{A} \& \overline{B}$	$\overline{(A \& B)} = \overline{A} \oplus \overline{B}$
$\overline{1} = \bot$	$\overline{\bot} = 1$
$\overline{0} = \top$	$\overline{\top} = 0$
$\overline{(!A)} = ?(\overline{A})$	$\overline{(?A)} = !(\overline{A})$

its linear negative, as such connectives can also be classified into positive ($\oplus$, $\otimes$, 1, 0, !) and negative ($\parr$, &, $\bot$, $\top$, ?). Below is a table of connectives and their respective duals.

Linear implication ($\multimap$) is not included in the table above as it doesn't have a dual *per se* although because of the tight relationship between duality and classical negation we can reason about it and derive it's dual. While in propositional logic we have the equivalency $A \Rightarrow B \equiv \neg A \vee B$ in linear logic we can use multiplicative disjunction and linear negation to achieve the same result and derive the following equivalency $A \multimap B \equiv \overline{A} \parr B$ [9]. Also of note is the obvious translation of De Morgan's Laws from classical propositional negation to linear negation, and as a consequence into duality.

2.2 Session Types

To properly understand what Session Types are, we first need to understand what a Type System is. A *Type System*, or *Typing Discipline*, is a mechanism built to guarantee the correctness of programs by assigning *types* to variables and functions, based on a defined set of typing rules, thus enforcing constraints on the behaviour and interactions of those functions and the domains of those variables. Any expression that can be successfully assigned a type by the application of one or more typing rules is considered a *well-typed term*. Type Systems, in a way, act as a filter that narrows the universe of possible programs in a language to those that are well-typed according to a given Type System. This restriction is necessary to eliminate typing errors and ensure the correctness of programs.

However, this "type security" comes with an associated cost of expressiveness. Some programs are rejected by the Type System because they are simply not typeable, a clear example of this is the *Y-combinator* of the λ-calculus ($Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$), although this function is useful to introduce recursion it is for that very same reason that it cannot be typed, as its type would need to be defined with respect to the itself. In other cases the functions are typeable, but the Type system is just too conservative, and as such rejects these functions as invalid. A simple example would be the program *[(if*

false then true else 0) + 2], which would be rejected because the conditional expression has branches of different types (*boolean* and *integer*), and as such is "ill-typed". This could be a valid program in a language that defines the types of its functions *dynamically* and that interprets *True* and *False* as numbers (e.g. Python). Here the conditional expression would evaluate to either "True" or "0", and then the addition operation would execute interpreting the value correctly ("1" and "0" respectively).

Traditionally the focus of the work on typing disciplines has been on *what* the result of a computation should be, and *what* it should look like [19]. But with the rise of concurrent and distributed systems, also arose the possibility to describe properties associated with the behaviour of such programs, meaning the focus of the study of type systems began to shift to the study of *how* a program reached the result. These "new" type systems are often referred to as *behavioural* systems, and Session Types are a part of such systems.

Session Types define the structure of communication between multiple processes by specifying the order of operations (choices, forks, and data operations) and the types of data being manipulated. They establish a clear protocol (or *session*) that designated channels must follow, ensuring that no typing errors occur during the communication [17]. This guarantees that a session-typed language enjoys the properties of type-safety and session fidelity, progress and some liveness properties. But in order to to allow a flexible description of communication structures, sessions aren't sufficient, we must also define three *communication primitives* [17]:

- **Value Passing**: The standard synchronous method of message-passing as described in [16].
- **Label Branching**: An alternative way of method invocation without value passing.
- **Delegation**: A way to pass along a channel to different processes.

For a better understanding of session type systems, we present below the syntax for typing the sessions offered by processes proposed in [17] and the syntax for the typing discipline, also defined in [17], below that.

A short explanation of the syntax as defined in [17] follows. S represents the *sorts*, which are similar to primitive data types, while α represents the type of interaction occurring in a channel. Of note, is the sort $\langle \alpha, \bar{\alpha} \rangle$ which represents two complementary (dual) structures of interaction that are associated with a name, in other words this syntactic construct represents the pair of *request-accept* operations associated with a name (for instance *a* in *accept* ***a****(k) in P*) as shown in Figure 2.2. The arrows $(\downarrow;\uparrow)$ represent input and output respectively of either values of sort S ($\downarrow [S];\alpha$ or $\uparrow [S];\alpha$), or channels ($\downarrow [\alpha];\beta$ or $\uparrow [\alpha];\beta$), note that these constructs also mean that after the output/input operation is made we continue with the another behaviour, either α ($\downarrow [S];\alpha$ or $\uparrow [S];\alpha$) or β ($\downarrow [\alpha];\beta$ or $\uparrow [\alpha];\beta$). $\&\{l_1 : \alpha_1, ..., l_n : \alpha_n\}$ denotes branching, where the process waits for a label l_i and then proceeds as α_i. While $\oplus\{l_1 : \alpha_1, ..., l_n : \alpha_n\}$ denotes the dual operation of sending label

P, Q	::=	request a(k) in P	session request
	|	accept a(k) in P	accepting session
	|	k![e_1...e_n];P	send data
	|	k?(x_1...x_n);P	receive data
	|	$k \triangleleft l$;P	selection
	|	$k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\}$	branching
	|	throw k[k'];P	send channel
	|	catch k(k') in P	receive channel
	|	if e then P else Q	conditional branching
	|	P | Q	parallel composition
	|	inact	inaction
	|	(vu)P	name/channel hiding
	|	def D in P	recursion
	|	X[ek]	process variables
D	::=	$X_1(x_1k_1) = P_1$ and ... and $X_n(x_nk_n) = P_n$	declaration for recursion

Figure 2.1: Syntax of Session Type Systems

$$S ::= \mathbf{nat} \mid \mathbf{bool} \mid \langle \alpha, \bar{\alpha} \rangle \mid \mathbf{s} \mid \mu s.S$$

$$\alpha ::= \downarrow [S]; \alpha \mid \downarrow [\alpha]; \beta \mid \&\{l_1 : \alpha_1, ..., l_n : \alpha_n\} \mid \mathbf{1} \mid \bot \\ \mid \uparrow [S]; \alpha \mid \uparrow [\alpha]; \beta \mid \oplus\{l_1 : \alpha_1, ..., l_n : \alpha_n\} \mid \mathbf{t} \mid \mu \mathbf{t}.\alpha$$

Figure 2.2: Type Syntax of Session Types

l_i and then proceeding as α_i. The impossibility of communication is represented by $\bot$, while inaction is represented by *1*. Finally recursion is represented by the construct $\mu t.\alpha$ and can be interpreted as "recursively act as α until t is satisfied".

In this next section we go a bit more *in depth* on some of the syntactic constructs to further clarify some concepts that might not be immediately clear from the above Figure 2.2, followed by another section with some examples of processes interacting and demonstrating the behaviour of some of the constructs defined above.

Session Creation - With respect to *request a(k) in P* and *accept a(k) in P* the former construct requests, through name *a*, that a session be started and a new channel *k* associated to said new session, then process P is free to use *k* in any valid way. The latter concept is a dual of this procedure, and so accepts a request for the start of a session via name *a* and creates a new channel *k* as requested, so that later process P can use it. This behaviour is

exemplified in Example 1.

Value Passing - Both *k![e];P* and *k?[x];P* are operations to communicate data between processes, the first construct signifies the sending of expression e over channel k, and then proceeds as process P. The second construct is the dual operation, and as such receives data through channel k and then binds the values of the received data to some variable x in P. It is of note that, as represented in the Figure 2.2 above, there can be multiple distinct expressions sent ($e_1 ... e_n$) and received ($x_1 ... x_n$) through a channel.

Branching - The operations of label branching ($k \triangleleft l;P$ and $k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\}$) are once again duals of each other, and while the first is used to send a label l through channel k and then continues as process P, the second one is used to receive a label l_i through channel k and then continue as the corresponding process P_i, labels $l_1 ... l_i$ are assumed to be pairwise distinct.

Another form of branching is conditional branching (*if e then P else Q*), where a process moves on as process P, if the boolean expression e evaluates as **True**, and as process Q otherwise.

Channel Forwarding - Channel forwarding is achieved by the *throw k[k'];P* and *catch k(k');P* constructs. The process sending a channel k' through channel k and then moving on as process P is described by the first construct, while the second one (being its dual) receives a channel through channel k and then binds it to k' in process P, through which it continues operating.

Scope Restriction - Codified by the construct *(vu)P*, and signifies that variable u is now bound (or hidden) to process P. It is also of note that *(vuu')P* is a valid use of this construct and not only binds both u and u' to P, but also establishes a "connection" between u and u', meaning that communication between two threads belonging to P (one which writes on u and another that reads from u', or vice-versa) can also be described by this construct. Also an important distinction is the fact that *(vuu')P* is neither equivalent, nor short form for *(vu)(vu')P*, as described in [24].

The following section provides some examples of theoretical processes interacting with each other through some of these constructs.

2.2.1 Examples

1. **Session Creation and Simple Value Passing:**

P ::= request a(k) in k![55906];inact

Q ::= accept a(k) in k?(x);inact

In this example process P requests the start of a session through channel a and then sends a value through that session's channel (name k). Process Q answers and accepts the session initiation request and then proceeds to receive a value through the new session's channel (name k), binding the received value to variable x. Observe

that these processes do not utilize the value that was passed for anything, as they become inactive right after sending/receiving the value this is just an example on how session creation and value passing could be achieved.

2. **Parallel Execution and Simple Branching:** For clarity in this example we first start by defining two processes P and Q:

$$\text{P ::= request a(k) in } k \triangleleft l_1\text{; k![55906]; inact}$$
$$\text{Q ::= accept a(k) in } k \triangleright \{l_1 : k?(x); inact | l_2 : k?(s); inact\}$$

The description of these processes' behaviour:

$$\text{P | Q}$$

This example introduces some complexity with the branching and parallel execution constructs, alongside the already established value passing and session creation from Example 1. The actual description of the processes' behaviour can be confusing and so it has been rearranged, but for completeness, here is the full description:

$$\text{request a(k) in } k \triangleleft l_1\text{; k![55906]; inact | accept a(k) in } k \triangleright \{l_1 : k?(x); inact | l_2 : k?(s); inact\}$$

In this example process P requests a new session, which Q accepts, then proceeds to (in order) send the label l_1 through channel k, and an integer also through channel k. Process Q receives the label first, forcing it to branch into receiving a value (in this case an integer) and binding it to x . If the label l_2 was sent, then the value sent/received through k would be a string, and it would've been bound to s instead of x.

3. **Simple Recursion:**

$$\text{P ::= request a(k) in def R(x, k) = (k![x]; } k \triangleright \{cont : R(x+1, k) | stop : inact\} \text{) in R(0, k)}$$

$$\text{Q ::= accept a(k) in def L(k) = (k?(x); if (x >= 3) then } k \triangleleft stop\text{; inact else } k \triangleleft cont\text{; L(k)) in L(k)}$$

In this example we have two processes that communicate between them a set number of times before stopping. Process P requests a new session, and then proceeds to define the recursive function *R(x,k)* in which we first send the value of x through channel k, which starts at 0 (*R(**0**,k)*) and then offer a choice on the same channel between continuing with the recursion and incrementing the value of x (*cont : R(x + 1, k)*) or stopping (*stop : inact*). Process Q starts by accepting the new session request, then proceeds to define a recursive function *(L(k))* that receives a value sent through

k and binds it to a variable *x*. It then tests if the value of *x* is greater or equal to 3 and branches accordingly, if the expression evaluates to **True** then Q sends the label *stop* through *k* and becomes inactive, otherwise, if the value of *x* is strictly less than 3, process Q sends the label *cont* through channel *k* instead and calls the recursive function *L(k)* again. This program ensures that the two processes communicate with each other twice before stopping, and while the processes are not communicating productively and no result is achieved from such communication, one can easily imagine how that would be possible.

2.3 Logical Session Types

Although Session Types can guarantee various beneficial properties for the communication of two processes, the problem of deadlocking still persists. To solve this Logical Session Types (LST) were introduced in the works of L. Caires and F. Pfenning [8] as a way to interpret Linear Logic propositions as communication protocols, thus taking advantage of the intrinsic characteristics of Linear Logic. In this "Propositions as Sessions" [25] paradigm processes correspond to proofs, process reduction/congruence corresponds to proof simplification through the cut rule, and as mentioned before Session Types correspond to propositions. This paradigm allows us to take advantage of the guarantees of progress, liveness, type safety and session fidelity properties intrinsic of Session Types, while also providing us with deadlock-freedom.

Generally Logical Session Types are implemented using the Intuitionistic interpretation of Linear Logic, due to its similarities with functional language and communication processes. However this work describes an implementation based on CLL because this allows us to later integrate some useful constructs like Shared Mutable State [22, 21], and allows us to model processes with multiple concurrent outputs.

Below we present the inference rules of a simple language based on LST, as described in [22, 21], as well as give a short explanation of each connective. We also present the syntax of *Types*, which is equivalent to the syntax of Linear Logic described in Section 2.1, and as such will have no explanation here. It is important to note that the language syntax presented here is a derivation from the CLASS language presented in [22] where we exclude the rules for Affinity, Polymorphism, Induction and Coinduction.

Types A, B are defined by:

$$\text{A, B} ::= \text{X} \mid 1 \mid \bot \mid A\&B \mid A \oplus B \mid A \parr B \mid A \otimes B \mid !\text{A} \mid ?\text{A}$$

Figure 2.3: Type Syntax

As duality is an integral part of Linear Logic it is also transferred into the syntax of LST, and is defined by:

$$\overline{1} = \bot \qquad \overline{!A} = ?\overline{B} \qquad \overline{A \otimes B} = \overline{A} \parr \overline{B} \qquad \overline{A \oplus B} = \overline{A} \& \overline{B}$$

Figure 2.4: Duality Associations

Typing Judgments: Typing judgments allow us to specify the types of channel endpoints, and they have the following form:

$$P \vdash x_1 : A_1, ..., x_n : A_n; y_1 : B_1, ..., y_m : B_m$$

They generally mean that a process P, offers the channel endpoints $x_1, ..., x_n, y_1, ..., y_m$, and specifies that these endpoints will follow protocols defined by their associated types $(A_1, ..., A_n, B_1, ..., B_m)$. It is important to distinguish between the two different types of contexts present in LST, the linear context Δ, which is linearly handled meaning we cannot discard or copy channels, and the exponential (or unrestricted) context Γ, which allows *weakening* and *contraction*. This way of distinguishing is based on [8] and assumes a pairwise distinction of channel names between the two contexts.

Composition: The *interactive composition* is the first process composition of LST, and is described by the cut rule

$$\frac{P \vdash \Delta', x : A; \Gamma \quad Q \vdash \Delta, x : \overline{A}; \Gamma}{P \,|x : A|\, Q \vdash \Delta' \Delta; \Gamma} \qquad \text{(Tcut)}$$

can be interpreted as two processes P and Q running concurrently and interacting along a private session x, P providing a behaviour of type A and Q receiving the dual of that behaviour $(\overline{A})$.

The other process composition of LST is the parallel composition, described by the rule

$$\frac{P \vdash \Delta'; \Gamma \quad Q \vdash \Delta; \Gamma}{P \,||\, Q \vdash \Delta', \Delta; \Gamma} \qquad \text{(Tpar)}$$

can be interpreted as two processes P and Q running in parallel, without ever interfering with each other, while offering a common interface.

Inaction and Termination: Inaction is the operational representation of doing nothing and is defined by the rule:

$$\frac{}{0 \vdash \emptyset; \Gamma} \qquad \text{(T0)}$$

Session termination is represented by the dual types *1* and $\bot$ and are defined by rules

$$\frac{}{\mathsf{close}\, x \vdash x : 1; \Gamma} \qquad \text{(T1)} \qquad \frac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\, x; Q \vdash \Delta, x : \bot; \Gamma} \qquad \text{(T}\bot\text{)}$$

where process *close x* closes/terminates a session, and process *wait x;Q* waits for the session to be closed, and continues as Q. This interaction can be modelled by the reduction rule:

$$\mathsf{close}\, x \,|x : 1|\, \mathsf{wait}\, x; Q \rightarrow Q \qquad (1\bot)$$

Forwarding: Operationally, forwarding represents the redirection of traffic between x and y, meaning that all interactions are delegated from y to x, and is represented by the rule:

$$\frac{}{\mathsf{fwd}\, x\, y \vdash x : \overline{A}, y : A; \Gamma} \quad \text{(Tfwd)}$$

The associated reduction is implemented by name substitution:

$$\mathsf{fwd}\, x\, y \,|y : A|\, P \rightarrow \{x/y\}P \quad \text{(fwd)}$$

Session Communication: Session communication is typed by the multiplicative connectives of Linear Logic ($A \otimes B$ and $\overline{A} \parr \overline{B}$) and is represented by the rules:

$$\frac{P_1 \vdash \Delta_1, y : A; \Gamma \quad P_2 \vdash \Delta_2, x : B; \Gamma}{\mathsf{send}\, x(y.P_2); P_1 \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma} \quad (\mathrm{T}\otimes)$$

$$\frac{Q \vdash \Delta, z : \overline{A}, x : \overline{B}; \Gamma}{\mathsf{recv}\, x(z); Q \vdash \Delta, x : \overline{A} \parr \overline{B}; \Gamma} \quad (\mathrm{T}\parr)$$

Process *send x(y.P_1);P_2* sends a fresh channel on session x. This process is composed of two independent parts, P_1 which implements and binds the new session y, and process P_2 which provides continuation on channel x. The dual process *recv x(z);Q* receives a name z on x and binds it to and continues as Q. As in Linear Logic $A \multimap B \triangleq \overline{A} \parr B$. The reduction rule that represents session communication is:

$$\mathsf{send}\, x(y.P_2); P_1 \,|x : A \otimes B|\, \mathsf{recv}\, x(z); Q \rightarrow P_2 \,|x : B|\, (P_1 \,|y : A|\, \{y/z\}Q) \quad (\otimes\parr)$$

As described before session $A \otimes B$ is *composed* of two parts, and this fact becomes evident in the reduction, where the cut of $A \otimes B$ gives way to two other cuts on sessions A and B, where the inner cut connects process Q with process P_1 and the outer cut connects process Q with process P_2, which is the continuation of the sending process. Another important observation is the evolution of the type of channel x, from $A \otimes B$ to B.

As a rule only new (bound) names can be sent in communication as specified in [8]. However we can encode the free output of unbounded channel y as $\mathsf{send}\, x\ y; P \triangleq \mathsf{send}\, x(z.\ \mathsf{fwd}\, y\, z); P$ without breaking any rules. The following typing rule is by consequence admissible:

$$\frac{P \vdash \Delta, x : B; \Gamma}{\mathsf{send}\, x\ y; P \vdash \Delta, y : \overline{A}, x : A \otimes B; \Gamma} \quad (\mathrm{T}\otimes_f)$$

Branching: the types of *offer* and *choice* are typed by the dual types $A\&B$ and $\overline{A} \oplus \overline{B}$, and are represented by the following set of rules:

$$\frac{P_1 \vdash \Delta, x : A; \Gamma \quad P_2 \vdash \Delta, x : B; \Gamma}{\mathsf{case}\, x\{|inl : P_1 \,|inr : P_2\} \vdash \Delta, x : A\&B; \Gamma} \quad (\mathrm{T}\&)$$

$$\frac{Q_1 \vdash \Delta', x : \overline{A}; \Gamma}{x.inl; Q_1 \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma} \quad (\mathrm{T}\oplus_l) \qquad \frac{Q_2 \vdash \Delta', x : \overline{B}; \Gamma}{x.inr; Q_2 \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma} \quad (\mathrm{T}\oplus_r)$$

The process $case\ x\{|inl : P_1\ |inr : P_2\}$ offers a menu of two choices, left (P_1) and right (P_2), and processes $x.inl; Q_1$ and $x.inr; Q_2$ are its duals, respectively making a choice between left and right. The associated reductions are:

$$\mathsf{case}\, x\{|inl : P_1\ |inr : P_2\}\ |x : A\&B|\ x.inl; Q_1 \rightarrow P_1\ |x : A|\ Q_1 \qquad (\&\oplus_l)$$

$$\mathsf{case}\, x\{|inl : P_1\ |inr : P_2\}\ |x : A\&B|\ x.inr; Q_2 \rightarrow P_2\ |x : B|\ Q_2 \qquad (\&\oplus_r)$$

An important encoding that can be extracted from this *offer-choice* duality is the definition of a *Linear Boolean*, with session type $Bool \triangleq 1 \oplus 1$ encoded as such:

$$|false\ b \vdash b : Bool \quad |true\ b \vdash b : Bool \quad |false\ b \triangleq b.inl; \mathsf{close}\, b \quad |true\ b \triangleq b.inr; \mathsf{close}\, b$$

Servers: Server definition and client invocation are typed by the exponential connectives of Linear Logic (!A an $?\overline{A}$), and are represented by the following rules:

$$\frac{P \vdash y : A; \Gamma}{!x(y); P \vdash x :!A; \Gamma} \quad \text{(T!)} \qquad \frac{Q \vdash \Delta; \Gamma, x : \overline{A}}{?x; Q \vdash \Delta, x :?\overline{A}; \Gamma} \quad \text{(T?)}$$

Process *!x(y);P* continuously offers on channel x a service A provided by P which is comparable to a server body. Process *?x;Q* simply moves a session x of type x: $?\overline{A}$ from the linear context to the unrestricted context, and proceeds as Q. Along with the rules for the server definition and the client invocation, we need to define a new cut rule that can act on the unrestricted context Γ, and a *call* function in order to call a service on a server:

$$\frac{P \vdash y : A; \Gamma \quad Q \vdash \Delta; \Gamma, x : \overline{A}}{y.P\ |!x : A|\ Q \vdash \Delta; \Gamma} \quad \text{(Tcut!)} \qquad \frac{Q \vdash \Delta, z : \overline{A}; \Gamma, x : \overline{A}}{\mathsf{call}\, x(z); Q \vdash \Delta; \Gamma, x : \overline{A}} \quad \text{(Tcall)}$$

The term $y.P\ |!x: A|\ Q$, can be interpreted as a server body P with a pool of clients Q, which can request a service zero or more times. As mentioned before process *call x(z);Q* calls a service on the server linked to x, that is to be processed by the newly-bounded name z, and then proceeds as Q. The reduction that models the behaviour between (T!) and (T?) is:

$$!x(y); P\ |x :!A|\ ?x; Q \rightarrow y.P\ |!x : A|\ Q \qquad (!?)$$

Notice that a linear cut on *x: !A* reduces to an unrestricted cut on *!x: A*, and it can be understood as the activation of the resources in the server's thread pool, so that clients may invoke them. Server invocation by clients is then modelled by:

$$y.P\ |!x : A|\ \mathsf{call}\, x(z); Q \rightarrow \{z/y\}P\ |z : A|\ (y.P\ |!x : A|\ Q) \qquad (\text{call})$$

A replica of the server's body in instantiated on the fresh channel z through the use of *z/yP*. Also of note is the availability of the server's body in the continuation process Q, making it possible to make further requests.

Finally, forwarding over the unrestricted context is modelled by the process:

$$\mathsf{fwd}\, w\, y_! \triangleq !w(z); \mathsf{call}\, y(z'); \mathsf{fwd}\, z\, z' \vdash w :!A; \Gamma, y : \overline{A}$$

2.4 Related works

2.4.1 The FreeST Language

The *FreeST* language, initially developed by B. Almeida, A. Mordido and V. Vasconcelos in [4] inspired by the research of P. Thiemann and V. Vasconcelos in [23], is a context-free session-typed functional language with a type system based on *System F* extended with session types. It leverages the use of linear types combined with session types to, similarly to our language, provide safe communication protocols in concurrent programs. Unlike our language the session types are not augmented with linear logic connectives. Instead *FreeST* focuses on extending the expressiveness of session types from regular protocols to context-free protocols, allowing for richer, possibly nested recursive communication patterns.

Utilizing context-free session types also allows the language to "let go" of the "shackles of tail recursion," as described by the authors. Under traditional session type systems, protocols are typically restricted to regular communication patterns, which limits recursion to tail-recursive structures. By extending session types so that their communication traces (i.e., sequences of actions) correspond to context-free languages, *FreeST* supports protocols that exhibit a non-tail recursive structure, allowing more expressive communication patterns than those permitted by standard session types.

Communication in *FreeST* is implemented through synchronous message-passing linear channels, where, similarly to our language, channels have two endpoints, each holding a dual session type. This ensures exact matching of communication, single-use channels that guarantee process termination, and exclusive ownership of channel endpoints, avoiding many concurrency races. However, *FreeST* does not guarantee deadlock freedom, whereas our use of cut-elimination from linear logic allows us to model communication in a way that ensures deadlock absence.

Like our own language, *FreeST* is compiled, with Haskell as its target language, because it offers built-in concurrency and strong typing, allowing the authors to focus on session-type design rather than low-level code generation. These considerations are similar to our own reasons for choosing to compile to *Go*.

2.4.2 The CLASS Language

The topic of session types has been extensively studied throughout the years, and from this work many implementations of session types in minimalist languages have been done. Generally these languages are essentially academic, aiming instead to refine the implementation of a type-checker and push the boundaries of what is possible. CLASS is one such language, developed by P. Rocha and L. Caires and presented in [22]. As our language will be largely based on CLASS here we introduce the constructs of this language not included in the previous section, and expand the syntax presented before, to match that of the original language (CLASS).

Types A, B of CLASS are defined by:

$$\begin{array}{lllllll} A, B ::= X & |1 & |\bot & |A\&B & |A \oplus B & |A \parr B & |A \otimes B \\ & |!A & |?A & |\exists X.A & |\forall X.A & |\mu X.A & |\nu X.A \\ & |\wedge A & |\vee A & |S_\bullet A & |S_\circ A & |U_\bullet A & |U_\circ A \end{array}$$

Figure 2.5: CLASS' Type Syntax

The types in the first two rows correspond to the types of *Second-Order Classical Linear Logic* (CLL incorporating second-order quantification), extended with the inductive and coinductive types (μ, ν). As before, types can represent variables (X), units (1, $\bot$), multiplicative ($\otimes$, $\parr$) and additive connectives (&, $\oplus$), exponentials (!, ?) and quantifiers ($\exists$, $\forall$). All other connectives have been explained in Section 2.1, apart from the quantifiers which are used to encode polymorphism as defined in [10]. The third row extends types with affinity ($\wedge$, $\vee$) and new modalities to type the *shared affine state* ($S_\bullet A$, $U_\bullet A$, $S_\circ A$, $U_\circ A$).

As mentioned previously, duality plays a major role in CLASS by capturing the symmetry in process interaction, as manifest in the *cut* rule. Duality in CLASS (definition presented below) corresponds to Linear negation, and is an involutive process as explained in Section 2.1.

Duality in CLASS is defined by:

$$\begin{array}{lll} \overline{1} = \bot & \overline{A \otimes B} = \overline{A} \parr \overline{B} & \overline{A \oplus B} = \overline{A}\&\overline{B} \\ \overline{!A} = ?\overline{B} & \overline{\exists X.A} = \forall X.\overline{A} & \overline{\mu X.A} = \nu X.\{\overline{X}/X\}(\overline{A}) \\ \overline{\wedge A} = \vee\overline{A} & \overline{S_\bullet A} = U_\bullet\overline{A} & \overline{S_\circ A} = U_\circ\overline{A} \end{array}$$

Figure 2.6: CLASS' Duality Definition

As mentioned before, the syntax and behaviour of the constructs and connectives described in Section 2.3 is a shortened version of CLASS' syntax and construct behaviour, as such we will only highlight what was left out in that Section. It is important to note that although the dyadic typing context ($\Delta;\Gamma$) remains the same, a new form of typing judgement arises to better suit the needs of coinduction. Typing judgments now have the form $P \vdash_\eta \Delta;\Gamma$, where the index η represents a finite map of coinductive hypothesis, in order to type corecursive processes, as detailed later.

Polymorphism: The existential connectives ($\exists X.A$, $\forall X.\overline{A}$), are the dual pair that describes type abstraction, and are defined by the following inference rules:

$$\frac{P \vdash_\eta \Delta, x : \{B/X\}A;\Gamma}{\mathsf{sendty}\ x(B); P \vdash_\eta \Delta, x : \exists X.A;\Gamma} \quad (\mathrm{T}\exists)$$

$$\frac{Q \vdash_\eta \Delta, x : \overline{A}; \Gamma}{\mathsf{recvty}\ x(X); Q \vdash_\eta \Delta, x : \forall X.\overline{A}; \Gamma} \quad (\mathrm{T}\forall)$$

In rule (T∀), notice that X does not occur free in the contexts Δ, Γ. Process *sendty x(B);P* sends the representation type B along x, and then continues as P. Inversely, process *recvty x(X);Q* receives that type information through x, and then proceeds as Q.

The associated reduction rule is:

$$\mathsf{sendty}\ x(B); P \mid x : \exists X.A \mid \mathsf{recvty}\ x(X); Q \longrightarrow P \mid x : \{B/X\}A \mid \{B/X\}Q \quad (\exists\forall)$$

An important illustration is the definition of inductive types through the use of these type quantifiers, shown here through the implementation of the natural with polymorphic sessions [18]:

$$Nat \triangleq \forall X.X \multimap !(X \multimap X) \multimap X$$

$$zero\ n \triangleq \mathsf{recvty}\ n(X); \mathsf{recv}\ n(z); \mathsf{recv}\ n(s); ?s; \mathsf{fwd}\ z\ n$$

$$succ\ n\ m \triangleq \mathsf{recvty}\ m(X); \mathsf{recv}\ m(z); \mathsf{recv}\ m(s); ?s; \mathsf{sendty}\ n(X); \mathsf{send}\ n\ z; \mathsf{send}\ n\ s; \mathsf{call}\ s(c); \mathsf{send}\ c\ n; \mathsf{fwd}\ c\ m$$

Terms of type *Nat* receive a type variable (X), a value (z: X) and a server (s: !(X ⊸ X)), then call the server a finite amount of times (possibly zero) to return a value (n: X). Noticeably, *zero n* ⊢ *n: Nat* simply forwards z on n, without a server call, thus representing the value of "0". However, *succ n m* ⊢ *n:* $\overline{Nat}$*, m: Nat* calls the server one extra time on the output produced by the calls of n, thus representing the *successor* function that returns the successor of a value.

Corecursiveness: Coinductive types are used to type corecursive processes, and have the following typing rules:

$$\frac{P \vdash_(\eta')\Delta, z : A; \Gamma \quad \eta' = \eta, X(z,w) \mapsto \Delta, z : Y; \Gamma}{\mathsf{corec}\ X(z,w); P[x,y] \vdash_\eta \{y/w\}\Delta, x : \nu Y.A; \{y/w\}\Gamma} \quad (\mathrm{Tcorec})$$

$$\frac{\eta = \eta', X(x,y) \mapsto \Delta, x : Y; \Gamma}{X(z,w) \vdash_\eta \{w/y\}\Delta, z : Y; \{w/y\}\Gamma} \quad (\mathrm{Tvar})$$

$$\frac{P \vdash_\eta \Delta, x : \{\mu X.A/X\}A; \Gamma}{\mathsf{unfold}_\mu\ x; P \vdash_\eta \Delta, x : \mu X.A; \Gamma}\ (\mathrm{T}\mu) \quad \frac{P \vdash_\eta \Delta, x : \{\nu X.A/X\}A; \Gamma}{\mathsf{unfold}_\mu\ x; P \vdash_\eta \Delta, x : \nu X.A; \Gamma}\ (\mathrm{T}\nu)$$

The corecursive processes *corec X(z,w);P[x,y]*, with parameters z, w bound in P, are instantiated with arguments x, y which are free in the process. It is convention to always offer the coinductive behaviour of type $\nu Y.A$ in the first argument z of a corecursive process. Note that by definition in rule (Tcorec), in order to type body P of a corecursive process, the map η is extended with a *coinductive hypothesis* binding X,a process variable, to the typing context Δ,*z: Y*;Γ so that we can type the body of P (the corecursion) and later appeal to X (which intuitively represents P itself) and recover P's typing invariant.

Most importantly, the type variable Y is free only in *z: A*, which causes corecursive calls to always be applied to names z'. These names hereditarily descend from the initial corecursive argument z. This restricted freedom of type variable Y is a necessary condition of strong normalization and corresponds to the restriction of corecursive calls to "smaller" argument sessions of inductive types.

The rule (Tvar) types a corecursive call *X(z,w)* by looking up in the map η for the corresponding binding and renaming the parameters to the arguments of the call. Finally inductive and coinductive are explicitly unfolded with rules (Tμ) and (Tν), but usually the unfolding is omitted in order to simplify presentation, being instead written as *rec A = f(A)* (or *corec B = f(B)*) instead of $A = \mu X.f(X)$ (or $B = \nu X.f(X)$). Another simplification is the representation of corecursive process definition, being written as *Q(x,y) = f(Q(-))* instead of *Q(x,y) = corec X(z,w); f(X(-)) [x, y]*

Affinity: Affinity describes discardable linear resources and is modelled by the typing rules

$$\frac{P \vdash_\eta a : A, b : \vee B, c : U_\bullet C; \Gamma}{\mathsf{affine}_{(b,c)}\, a; P \vdash_\eta a : \wedge A, b : \vee B, c : U_\bullet C; \Gamma} \quad \text{(Taffine)}$$

$$\frac{}{\mathsf{discard}\, a \vdash_\eta a : \vee A; \Gamma} \quad \text{(Tdiscard)} \qquad \frac{Q \vdash_\eta \Delta, a : A; \Gamma}{\mathsf{use}\, a; Q \vdash_\eta \Delta, a : \vee A; \Gamma} \quad \text{(Tuse)}$$

Affine session are introduced by rule (Taffine) that promotes *a: A* to an affine session *a:* $\wedge A$, that can either be used or discarded. This rule types the process $\mathsf{affine}_{(b,c)}\, a; P$, which provides the affine session a and then continues as P. The promotion of a session can only happen if that session only depends on resources that can be disposed, as described by $b : \vee B, c : U_\bullet C; \Gamma$. What this means is that every resource on which a session depends, must satisfy some form of weakening, namely: coaffine session b_i of Type $\vee B_i$, full cell usages c_i of type $U_\bullet C_i$, and unrestricted sessions in the context Γ. Although often omitted in practice, the dependencies of an affine object on coaffine objects or full cell objects are explicitly annotated as *b,c* in the process term ($\mathsf{affine}_{(b,c)}$)

The coaffine endpoint $\vee A$ of an affine session has two operations: *use* (Tuse), and discard (Tdiscard). This endpoint has its dual in $\wedge\overline{A}$. The process represented by *use a;Q* uses a coaffine session a and continues as Q (dereliction rule). The process *discard a* discard a coaffine session a, serving as a weakening rule.

Shared Mutable State: Shared Mutable state is typed by rules:

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\mathsf{cell}\, c(a.P) \vdash_\eta \Delta, c : S_\bullet A; \Gamma}\ (\text{Tcell}) \qquad \frac{}{\mathsf{release}\, c \vdash_\eta c : U_\bullet A; \Gamma}\ (\text{Trelease})$$

$$\frac{}{\mathsf{empty}\, c \vdash_\eta c : S_\circ A; \Gamma}\ (\text{Tempty}) \qquad \frac{Q \vdash_\eta \Delta, a : \vee A, c : U_\circ A; \Gamma}{\mathsf{take}\, c(a); Q \vdash_\eta \Delta, c : U_\bullet A; \Gamma}\ (\text{Ttake})$$

$$\frac{Q_1 \vdash_\eta \Delta_1, a : \wedge \overline{A}; \Gamma \quad Q_2 \vdash_\eta \Delta_2, c : U_\bullet A; \Gamma}{\mathsf{put}\, c(a.Q_1); Q_2 \vdash_\eta \Delta_1, \Delta_2, c : U_\circ A; \Gamma}\ (\text{Tput})$$

Shared Mutable State is modelled by *mutex memory cells*, and its associate operations for client-use. At any moment a cell can be either *full* or *empty*. A full cell c, written as *cell c(a.P)* (typed $S_\bullet A$), stores an affine session a, of type $\wedge A$ defined by P. All objects stored by this cell are required to be affine so that we can safely dispose of memory cells without causing memory leaks. An empty cell is typed $S_\circ A$, and written as *empty c*.

Clients manipulate cells via the *take, put and release* operations. These operations apply to names of cell usage types $U_\bullet A$, $U_\circ A$ (full and empty respectively). A client can execute a take operation (*take c(a);Q*) if it owns a $U_\bullet A$-typed usage to a cell. This operation waits to acquire the cell mutex c, and then reads it into a. The cell becomes empty and the client process proceeds as Q. To release the lock, the taking process is attributed the responsibility of repopulating the cell with the use of the put operation (*put c(a.Q_1);Q_2*), where the stored object a (implemented by Q_1) is required to be affine.

The release operation allows a thread to drop its cell usage c. This release can only happen if the releasing thread is not holding the lock to the cell (i.e. the cell has the unlocked state, $U_\bullet A$). When all owning threads release their usages for a cell c, the cell is disposed, and the contents discarded. This eventually happens in well-typed programs.

In CLASS, multiple cell usages may be shared between concurrent threads, which compete to take and use the cell in interleaved critical sections. These usages can be passed around dynamically, changing the sharing topology at runtime.

The sharing of cell usages is expressed by the typing rules:

$$\frac{P \vdash_\eta \Delta', c : U_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : U_\bullet A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash_\eta \Delta', \Delta, c : U_\bullet A; \Gamma}\ (\text{Tsh})$$

$$\frac{P \vdash_\eta \Delta', c : U_\circ A; \Gamma \quad Q \vdash_\eta \Delta, c : U_\bullet A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash_\eta \Delta', \Delta, c : U_\circ A; \Gamma}\ (\text{TshL})$$

$$\frac{P \vdash_\eta \Delta', c : U_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : c : U_\circ A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash_\eta \Delta', \Delta, c : U_\circ A; \Gamma}\ (\text{TshR})$$

Co-contraction, introduced in Diferential Linear Logic (DiLL) [12] allows for finite multisets of linear resources to safely interact in cut-reduction, as such concurrent sharing

$$\mathsf{unfold}_{\mu}\, x; P \,|x|\, \mathsf{unfold}_{\nu}\, x; Q \rightarrow P \,|x|\, Q \qquad (\mu\nu)$$

$$\mathsf{unfold}_{\mu}\, x; P \,|x|\, \mathsf{corec}\, Y(z,w); Q[x,y] \rightarrow P \,|x|\, \{x/z\}\{y/w\}\{\mathsf{corec}\, Y(z,w); Q/Y\}Q \qquad (\text{corec})$$

$$\mathsf{affine}_{(b,c)}\, a; P \,|a|\, \mathsf{use}\, a; Q \rightarrow P \,|a|\, Q \qquad (\wedge \vee u)$$

$$\mathsf{affine}_{(b,c)}\, a; P \,|a|\, \mathsf{discard}\, a \rightarrow \mathsf{discard}\, a \,||\, \mathsf{release}\, c \qquad (\wedge \vee d)$$

$$\mathsf{cell}\, c(a.P) \,|c|\, \mathsf{release}\, c \rightarrow P \,|a|\, \mathsf{discard}\, a \qquad (S_{\bullet}U_{\bullet}r)$$

$$\mathsf{cell}\, c(a.P) \,|c|\, \mathsf{take}\, c(a'); Q \rightarrow P \,|a|\, (\mathsf{empty}\, c \,|c|\, \{a/a'\}Q) \qquad (S_{\bullet}U_{\bullet}t)$$

$$\mathsf{empty}\, c \,|c|\, \mathsf{put}\, c(a.P); Q \rightarrow \mathsf{cell}\, c(a.P) \,|c|\, Q \qquad (S_{\circ}U_{\circ})$$

$$P \leqslant P' and\ P' \rightarrow Q' and\ Q' \leqslant Q \supset P \rightarrow Q \qquad (\leqslant)$$

$$P \rightarrow Q \supset C[P] \rightarrow C[Q] \qquad (\text{cong})$$

Figure 2.7: Reduction rules for CLASS

becomes nondeterministic, which is required to soundly model memory cells and the linear concurrent usages. The construct *share c P || Q*, shares cell usage c with concurrent threads P and Q, by interpreting cocontraction. As opposed to cut, *share c P || Q* is not a binding operator, meaning the shared usage c is free in the conclusion of the typing rule (Tsh), allowing for an arbitrary amount of threads to share c, by iterating over nested uses of the typing rule.

While (Tsh) types the sharing of a full (and therefore unlocked) cell usage $U_{\bullet}A$, the symmetric rules (TshL) and (TshR) type sharing of an empty (and therefore locked) cell usage $U_{\circ}A$. We can verify that in a well-typed process at most one unguarded operation to c may use the type $U_{\circ}A$ while all others must use $U_{\bullet}A$. This establishes that, at runtime, only one thread may posses the lock for a given cell, which (by definition) is empty. The lock owning thread must execute a put operation on that cell, releasing its lock, while all other threads must be either attempting to *take* or *release* the reference to the cell. As a consequence of this behaviour we guarantee that no deadlock can occur and as such, despite all the additions to the language, CLASS can still ensure the correctness of programs and deadlock-freedom in communication.

To complete our short analysis of the CLASS we now present the reduction rules not already mentioned in Section 2.3, or here:

As a final note it is important to mention that not all languages implementing sessions types are interpreted languages, some are compiled languages like the aforementioned FreeST [3] and Concurrent C0 [26], as well as the language developed by J. Geraldo in [14], and this work.

3

Developed Language

The language that we developed in this work is a session-typed functional language, based on the CLASS language developed by P. Rocha and L. Caires in [22], and the language developed by J. Geraldo in his work [14]. These languages have their foundation on the *Proposition-as-Types* interpretation of Linear Logic, thus guaranteeing the absence of deadlocks, the termination of programs, and prevent the misuse or leakage of resources.

Our language builds on the foundation established by J. Geraldo, extending his framework for session-typed programming. Whereas Geraldo's language was grounded solely on an intuitionistic interpretation of linear logic, our work incorporates functionality for both intuitionistic and classical interpretations. This generalization preserves the functional programming features and channel-based concurrency as well as a strict separation between functional connectives and concurrency primitives of the original design. As in Geraldo's system, these primitives form a dedicated *process layer*. However, our contribution introduces the freedom to instantiate said *process layer* with either intuitionistic processes (as originally defined) or with the newly introduced classical processes, depending on the chosen interpretation.

In our system, terms in the *process* layer (which we will dub either *intuitionistic processes* or *classical processes*, for intuitionistic and classical terms respectively) retain unrestricted access to terms from the *functional* layer. By contrast, the functional layer's use of process terms remains carefully regulated, ensuring that interactions between the two layers respect the language's type discipline and preserve the compile-time correctness mentioned before.

3.1 Syntax

The syntax for the developed language is presented in Figures 3.1 and 3.2. Process terms are ranged over by P, Q, *Classical* processes by P_c, Q_c and functional terms by M, N. The channels are ranged over by c, d and variables by x, y while type variables are represented by T.

A program in our language is composed of a set of functional declarations, a set of type

declarations, which can include the definition of session-types, and a distinct, top-level declaration (dubbed *main*) which acts as the program's entry point. A declaration in our language has the form $x : T = N$, where the name x defined with type T can occur in expression N so as to allow for recursion when defining a function. To distinguish between a *Classical* process and an *Intuitionistic* process, we wrap the former in a *c_decl* block annotation.

$$\begin{aligned} M, N ::= {} & V \mid \texttt{fun}\ \overline{x} \to M\ \ \texttt{end} \mid \texttt{let}\ x = M\ \ \texttt{in}\ N \mid M\ N \\ & \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2\ \texttt{endif} \mid c \leftarrow \{P\} \leftarrow d_1 ... d_n \end{aligned}$$

Figure 3.1: Syntax of Expressions (M, N)

$$\begin{aligned} P, Q ::= {} & \texttt{send}\ c\ M; P \mid x : T \leftarrow \texttt{recv}\ c; P \mid \texttt{close}\ c \mid \texttt{wait}\ c; P \mid \texttt{fwd}\ c\ d \\ & \mid c \leftarrow \texttt{spawn}\ M\ d_1..d_n; P \mid \texttt{case}\ c\ \texttt{of}\ l_j : (P_j) \mid c.l; P \mid \texttt{sendc}\ c\ d; P \\ & \mid x : T \leftarrow \texttt{recvc}\ c; P \mid \texttt{print}\ c; P \mid \texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif} \end{aligned}$$

$$\begin{aligned} P_c, Q_c ::= {} & \texttt{send}\ c\ M; P \mid x : T \leftarrow \texttt{recv}\ c; P \mid \texttt{close}\ c \mid \texttt{wait}\ c; P \mid \texttt{fwd}\ c\ d \\ & \mid c, e_1..e_n \leftarrow \texttt{cspawn}\ M \leftarrow d_1..d_n; P \mid \texttt{case}\ c\ \texttt{of}\ l_j : (P_j) \mid c.l; P \\ & \mid \texttt{sendc}\ c\ d; P \mid x : T \leftarrow \texttt{recvc}\ c; P \\ & \mid \texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif} \mid \texttt{print}\ c; P \\ & \mid d \leftarrow \texttt{accept}\ c; P \mid affine_{[b,c]}(c); P \mid discard(c); P \\ & \mid use(c); P \mid \texttt{call}\ c(d); P \\ & \mid \texttt{empty}\ c : A_c \mid \texttt{cell}\ c(d)P \\ & \mid \texttt{take}\ c(d); P \\ & \mid \texttt{put}\ c(aQ); P \\ & \mid \texttt{release}\ x; P \mid \texttt{share}\ c\{P \mid\mid Q\} \\ & \mid \texttt{terminate} \end{aligned}$$

Figure 3.2: Syntax of Processes (P, Q) and Classical Processes (P_c, Q_c)

3.1.1 Functional Layer

The functional terms of our language form a conventional functional core, grounded in standard λ-calculus constructs, which we summarize here. Basic values (e.g., *int*, *bool*), binary operations (arithmetic such as +, - and relational, such as <, >, =), and unary operations (e.g. logical *not*) are collectively abstracted by the meta-variable V. Functions are written as multi-argument λ-abstractions of the form $\texttt{fun}\ x_1 ... x_n \to M\ \texttt{end}$, where type annotations for λ-bound resources may be omitted thanks to bi-directional typechecking (see Section 4.1). Recursive definitions are expressed with $\texttt{let}\ x = M\ \texttt{in}\ N$, and branching takes the familiar form $\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2\ \texttt{endif}$.

A central syntactic construct is `c` $\leftarrow$ `{P}` $\leftarrow d_1..d_n$, which encapsulates a process layer term as an *opaque* functional value. This construct does not evaluate P, instead, P offers some session behavior on channel c while consuming resources on channels $d_1..d_n$ (all bound in P). This mechanism enforces the intended control over which process layer terms may be accessed by the functional layer.

3.1.2 Process Layer

The session behaviour and resource usage is defined by the various process constructs, be they classical or intuitionistic, and while there are some important differences between these two interpretations' constructs they work in much the same way. The first difference is in the introduction of the *unrestricted* context Γ, which represents a collection of resources that can be reused, meaning they're no longer linear. This allows for the implementation of client/server functionality through the use of two new constructs (*accept* and *call*). The second difference is the reliance on classical duality. In classical processes every session type (A) is paired with a canonical dual $\overline{A}$, and a channel endpoint typed with A must always interact with a peer endpoint typed with $\overline{A}$. This makes communication protocols symmetrical and enforces precise alignment between the two sides of a channel. In contrast, intuitionistic processes do not rely on such a global notion of duality, instead capturing the behaviour at each endpoint directly through typing judgments.

For example, a process like `send` $c\ M; P$ denotes a send action through channel c of a previously evaluated functional term M, continuing with the behaviour defined by process P. Its dual, $x : T \leftarrow$ `recv` $c; P$ denotes that a process is waiting to receive a term (of type T) through channel c, which is then bound to variable x in the continuing process P. The communication of functional terms and channels works analogously, meaning that `sendc` $c\ d; P$ defines a process that sends channel d through channel c, then continues with the behaviour of P, and $x : T \leftarrow$ `recvc` $c; P$ defines a process that receives a channel with session-type T through c, then binds it to variable x in the continuation P. Apart from the duality distinction, all of the previously explained processes work in the same way for both classical and intuitionistic processes.

To signal the end of communication over a specific channel (c) we use the construct `close` c. If a process needs to await the closure of a session channel we write `wait` $c; P$, where we await a closing operation on the channel c before continuing operations as defined by process P. If we wish to produce a readable output from the result of a process we can make use of the `print` $M; P$ construct, which prints the result of M before continuing as P. A very useful operation is the redirection of inputs and outputs, done through the construct `fwd` $c\ d$. Note that while both channels must have the same type in the intuitionistic form, in the classical form they must be duals of each other.

It is commonplace for session-based languages to allow programs to branch, and in our case this can be done in one of two ways. The first (and simplest) construct that allows for a branching is the conditional (`if` M `then` N_1 `else` N_2 `endif`) construct,

which depending on the truth-value of M will either proceed as N_1 or N_2. The construct `case` c `of` $l_1 : (P_1) ... l_j : (P_j)$ denotes a choice, offering us the possibility of selection and the second form of branching. The process corresponding to this construct will await to receive a label l_j on channel c and then proceed with the behaviour defined by the associated process P_j. The process responsible for sending the label, makes use of the $c.l; P$ construct, which denotes the selection of a branch labelled l (this selection is communicated through channel c) and then continuing as specified in process P. In order to avoid miscommunications and possible errors, the typing system ensures that all possible branching options are covered by the choice construct.

The newly introduced constructs $d \leftarrow$ `accept` $c; P$ and `call` $c(d); P$ are unique to classical processes, because they rely on the unrestricted context Γ. The *call* construct can be viewed as a client which creates a new channel d and sends it through the (shared) channel c before continuing as P. The *accept* construct can be thought of as a server that awaits a connection (a receive channel operation on the same shared channel c) before binding the received channel to d and spawning a new process (running in parallel) following the behaviour of P. Intuitively, this can be seen as a server awaiting client requests, and then launching a new separate thread to handle the execution of each client in parallel. This explains why channel c must belong to the unrestricted context: it is shared and reused across many client-server interactions. Importantly, once the initial *call*/*accept* handshake is complete, each client-server pair communicates over its own fresh channel d, ensuring isolation and preventing deadlocks.

Also added in our language are the constructs representing promotion of a session to an affine session `affine` $(c); P$, and the operations associated `discard` $(c); P$ and `use` $(c); P$. These affine sessions represent discardable linear resources, which can either be used as a linear session or discarded. In order to allow a resource to be promoted to the affine state, we must first ensure that all session depending on said resource (its dependencies), are also *discardable*, meaning they are under some form of weakening rule (resources in the unrestricted context Γ, or other affine sessions), then we can use the aforementioned construct *affine* to promote the session. There are two possible interactions for an affine session, one is to *discard* it, meaning the session will be closed and removed from the linear context. The other one is to *use* it, meaning to *demote* the session back into a normal linear resource.

Finally the most interesting constructs added to the language, are the ones that allow for the modeling of shared mutable resources (memory cells), `cell` $c(d)P$, `empty` $c : A_c$, `take` $c(d); P$, `put` $c(aQ); P$, `release` $x; P$ and `share` $c\{P \mid\mid Q\}$. These cells can hold one session at a time, and can either be in an empty or full state. To create a full cell we make use of the *cell* construct, which allows us to promote a session into a cell and simultaneously place a session following the behaviour of P in the cell. An empty cell is created with the *empty* construct, and simply promotes a session to a cell. To interact with a full cell a session might utilize the *take* construct, effectively grabbing the stored session d, through c (the cell's channel), locking the cell, and then interacting with the process P, launched

in a new thread, through the grabbed session. If a cell is empty, some process holding the lock to the cell must execute a *put* operation, repopulating the cell, and unlocking access to it. Only one process may hold the lock to the cell at any one time. The *release* construct releases it's reference (channel c) to the cell, effectively closing communication on it's side. This can only happen if the cell is not locked. Finally, in order to allow multiple sessions to interact with the same cell, we can share a cell's reference (it's channel, c), by using *share*, this construct simply passes the channel into the linear construct of it's branches (processes P and Q).

3.1.3 Simulating Process Execution

The *spawn* (or *cspawn* for the classical version) construct is central to our language. It appears as $c \leftarrow \texttt{spawn}\ M\ \ d_1..d_n; P$ for an intuitionistic process and as $c, e_1..e_n \leftarrow \texttt{cspawn}\ M \leftarrow d_1..d_n; P$ for a classical process. Both variants allow processes to be used as values within other processes. The classical form, however, is overloaded to also manage the unrestricted context of a process in a way similar to the use of the *unrestricted cut rule* discussed in [21].

Operationally, the execution of $c \leftarrow \texttt{spawn}\ M\ \ d_1..d_n; P$ will (by type safety) evaluate M to an embedded process of the form $c \leftarrow \{Q\} \leftarrow d'_1..d'_n$. The resulting process Q then runs in parallel with the continuation P where channel c is then bound so it can be used as the communication link between Q and P. The list of channels $d_1..d_n$ (which can be seen as a context) supplies the free channels $d'_1..d'_n$ bound in Q.

As for the classical variant ($c, e_1..e_n \leftarrow \texttt{cspawn}\ M \leftarrow d_1..d_n; P$), the expression M likewise evaluates to a process $c \leftarrow \{Q\} \leftarrow d'_1..d'_n, e'_1..e'_n$, which runs in parallel with the continuation P. Here, channel c again serves as the communication link, and the linear context $d_1..d_n$ still supplies the free channels $d'_1..d'_n$ of Q. In addition, the classical form introduces an unrestricted context (reusable resources) $e'_1..e'_n$ which is instantiated by $e_1..e_n$.

A second key distinction concerns duality. Whereas the intuitionistic *spawn* uses c in Q directly, the classical *cspawn* dualizes it: in P the channel c has session-type A, while in Q it has the dual type $\overline{A}$.

For convenience, the syntax can be simplified in both variants: instead of writing M and requiring its evaluation to yield $c \leftarrow \{Q\} \leftarrow d'_1..d'_n$, one may write the embedded process directly as $\{Q\}$, leaving any contexts and channels implicit.

3.2 Typing

Our language's syntax of types has a basis on the one described in J. Geraldo's work [14], which is then extended with some constructs from the CLASS language as described in [22], resulting in the typing syntax presented in Figure 3.3. We use T,S to range over functional types, which assign types to functional terms, and A,B and A_c, B_c range over

session types and classical session types respectively, which specify the behaviour of *channels* in a process. For readability and reuse, session types can also be introduced as declarations (D), where a name variable N serves as an alias for the session type A.

$$D ::= \text{stype } N = A \mid \text{c\_stype } N = A$$
$$T, S ::= \text{Num} \mid \text{Bool} \mid \text{Unit} \mid T \to S \mid \{B_1 ... B_n \vdash A\} \mid \{B_1 ... B_n, C_1 ... C_n \vdash A\}$$

Figure 3.3: Functional Types (T, S) and Declarations (D)

In addition to the usual simple data types of a functional language (such as numbers and booleans) and function types $T \to S$, our language must also assign types to process terms. This is done using the types $\{B_1 ... B_n \vdash A\}$ and $\{B_1 ... B_n, C_1 ... C_n \vdash A\}$, which correspond to functional terms of the form $\mathsf{c} \leftarrow \{P\} \leftarrow d_1 .. d_n$ and $\mathsf{c} \leftarrow \{P\} \leftarrow d'_1 .. d'_n, e'_1 .. e'_n$, respectively. In both cases, the process P offers a session type A, while relying on the channels in its context typed by $B_1 ... B_n$ in the first form, and by $B_1 ... B_n$ together with $C_1 ... C_n$ in the second (classical) form.

3.2.1 Session Types

Session types describe channel behaviour as a structured sequence of communication actions. While our language distinguishes between classical and intuitionistic session types, they behave in much the same way. The key difference lies in how duality is treated: in classical processes, duality is explicit and appears directly in the typing through the expression $\overline{A_c}$, whereas in intuitionistic processes duality is only enforced indirectly by the typing judgments at each end of the channel.

For example, a channel of type $A \multimap B$ (formally $A \parr B$ or $A_c \parr B_c$ in the classical setting) represents a process that first **receives a channel** of type A and then continues as B. To interact with it, we need another process offering a channel typed with the dual session type $A \otimes B$, which instead **sends a channel** of type A before proceeding with B. Figure 3.5 summarizes the dual pairs of session types.

$$\begin{aligned} A_c, B_c ::= {} & A_c \parr B_c \mid A_c \otimes B_c \mid T \wedge B_c \mid T \supset B_c \mid \&\{l_1 : (A_1) ... l_i : (A_c)_i\} \\ & \mid \oplus \{l_1 : (A_1) ... l_i : (A_c)_i\} \mid \mu X.A_c \mid !A_c \mid ?A_c \mid \overline{A_c} \mid \wedge A_c \mid \vee A_c \\ & \mid S_\bullet A_c \mid S_\circ A_c \mid U_\bullet A_c \mid U_\circ A_c \mid X \mid N \mid 1 \mid \bot \\ A, B ::= {} & A \multimap B \mid A \otimes B \mid T \wedge B \mid T \supset B \mid \&\{l_1 : (A_1) ... l_i : (A_c)_i\} \\ & \mid \oplus \{l_1 : (A_1) ... l_i : (A_c)_i\} \mid \mu X.A \mid X \mid N \mid 1 \end{aligned}$$

Figure 3.4: Session Types (A, B) and Classical Session Types (A_c, B_c)

$$\begin{array}{rclcrclcrcl}
\overline{1} & \triangleq & \bot & & \overline{(A_c \otimes B_c)} & \triangleq & \overline{A_c} \parr \overline{B_c} & & \overline{(A_c \oplus B_c)} & \triangleq & \overline{A_c} \& \overline{B_c} \\
\overline{!A_c} & \triangleq & ?\overline{A_c} & & \overline{(T \wedge B_c)} & \triangleq & T \supset \overline{B_c} & & \overline{\wedge A_c} & \triangleq & \vee \overline{A_c} \\
\overline{S_\bullet A_c} & \triangleq & \overline{U_\bullet A_c} & & \overline{S_\circ A_c} & \triangleq & \overline{U_\circ A_c} & & & &
\end{array}$$

Figure 3.5: Duality of Types

We can also type the input and output of other data types using the types $T \supset B$ (concrete syntax of T => B) to input a functional value of type T and then continues to behave as defined by B. Dually $T \wedge B$ denotes output of functional value T and continuation type B. To type choice we have $\&\{l_1 : (A_1) ... l_i : (A_c)_i\}$ which denotes a channel whose offering process is waiting to receive one of the labels l_i listed, and then proceed with the behaviour defined by session type $(A_c)_i$ (or A_i for the intuitionistic form). To effectively make this choice we use type $\oplus\{l_1 : (A_1) ... l_i : (A_c)_i\}$ which types a channel that sends one of the available labels (l_i) and then has a continuation according to $(A_c)_i$ (or alternatively A_i). A closed (inactive) session is represented by *1*, a session waiting to be closed is represented by $\bot$ and N is used to represent a named type (or session type). Recursion is achieved through the type $\mu X.A$ (classical form $\mu X.A_c$) and the recursive type variable X.

Servers, affine states and shared memory cells are all specific to the classical version of our language and as such are defined solely by the classical session types $!A_c$, $?A_c$, $\wedge A$, $\vee A$, $S_\circ A$, $S_\bullet A$, $U_\circ A$ and $U_\bullet A$.

3.2.2 Typing Rules

In order to define typing rules we must first define the typing judgments, of which our language provides three. To type functional terms we utilize $\Psi \Vdash M : T$ which states that under the typing assumptions for free variable of form *x:T*, tracked in context Ψ, functional term M has type T. To type intuitionistic processes we utilize typing judgment $\Psi, \Delta \vdash P :: c : A$ which states that a process P will offer the behaviour of session type A along channel c, by using the resources specified by the *linear context* Δ and the functional environment (typing assumptions) Ψ. Finally, classical processes are typed using the judgment: $P \vdash \Psi; \Gamma; \Delta, c : A$, which means that process P provides a session of type A along channel c, while the contexts Ψ, Δ and Γ specify the resources and assumptions with which P interacts, both consuming and producing them in the style of classical linear logic.

Functional constructs can be typed by the following typing rules:

$$\frac{}{\Psi; x : T \Vdash x : T} \quad \text{(Var)}$$

This foundational rule simply states that variable x will have type T if such type T is attributed to the free variable x in the environment Ψ.

$$\frac{\Psi, x_1 : T_1 ... x_n : T_n \Vdash M : S}{\Psi \Vdash \texttt{fun}\ \ x_1 ... x_n \rightarrow Mend : T_1 ... T_n \rightarrow S} \quad \text{(Fun)}$$

The *Fun* rule defines a function, and provided M has type S in a context containing all $x_1 : T_1 ... x_n : T_n$ type assumptions, then the term `fun` $x_1 ... x_n \rightarrow$ `M end` will have type $T_1 ... T_n \rightarrow S$. This means that the function body M evaluates in a context where all function arguments $x_1 ... x_n$ are correctly associated to their types $\overline{T}$.

In order to make use of said functions, we need the **application** construct of the form `M N` as enumerated in 3.1 and typed by the rule:

$$\frac{\Psi \Vdash M : T_1 ... T_n \rightarrow S \qquad \Psi \Vdash N : T}{\Psi \Vdash MN : S} \quad \text{(FunApp)}$$

Here we state that given an expression M of type $T_1 ... T_n \rightarrow S$ (a function) and that term N has type T, then we can safely apply M to N with the application `M N` having type S.

$$\frac{\Psi \Vdash M : S \qquad \Psi, x : S \Vdash N : T}{\Psi \Vdash \texttt{let}\ \ x = M \texttt{in}\ \ N : T} \quad \text{(Let)}$$

With the above rule we can define let statements (`let` $x = M$ `in N`) which are typed as T provided that M is of type S and that in an environment where x has type T, N also presents that same type.

The constructs that allow processes (classical and intuitionistic) to be embedded into the functional layer are defined by the typing rules:

$$\frac{\Psi; d_1 ... d_n : B \vdash P :: c : A}{\Psi \vdash F\ c \leftarrow \{P\} \leftarrow d_1 : B_1 ... d_n : B_n \vdash A} \quad \text{(ProcExp)}$$

$$\frac{\Psi; d_1 ... d_n : B \vdash P :: c : A}{\Psi \vdash F\ c \leftarrow \{Q\} \leftarrow d'_1 : B_1 ... d'_n : B_n, e'_1 : C_1 ... e'_n : C_n \vdash A} \quad \text{(ClassProcExp)}$$

As discussed in Subsection 3.1.3, these constructs enable the functional layer to encapsulate processes, thereby providing the sole interface between the functional and process layers and thus ensuring the described control.

Across the two variants, the typing rules capture the same typing principle: the functional term `c` $\leftarrow$ `{P}` $\leftarrow d_1 ... d_n$ has type $\{B_1 ... B_n \vdash A\}$ whenever the process P is

well-typed in a context where it consumes channels $d_1...d_n$ of types $B_1...B_n$ and offers type A along channel c. The only difference arises in the classical variant (`ClassProcExp`), which additionally permits unrestricted channels, typed as $C_1...C_n$.

This implies that such values may appear within other processes, enabling processes to spawn or encapsulate further processes that then execute concurrently. The typing rule governing this behavior is:

$$\frac{\Psi \vdash M : \{B_1...B_n \vdash A\} \quad \Delta' = d_1 : B_1...d_n : B_n \quad P \vdash \Delta, c : A}{c \leftarrow \text{spawn } M\ d_1...d_n; P \vdash \Delta, \Delta', c : A} \tag{Spawn}$$

This rule states that the process expression Q (typed by $\{B_1...B_n \vdash A\}$) will be launched in parallel with process P, given that channels $d_1...d_n$ (typed $B_1...B_n$) are available and provided. In order to not break linearity we capture the channels in a new context region λ' meaning the channels will no longer be available for use in the *continuation process* P. The connection between these two processes is the channel c offered by Q and typed according to A. In our language the *cut-reduction* rule that simulates execution of a process is modelled by this *Spawn* rule.

A similar rule is used to type the *spawning* of classical processes:

$$\frac{\Psi \vdash M : \{B_1...B_n, E_1...E_n \vdash A\} \quad \Delta' = d_1 : B_1...d_n : B_n \quad P \vdash \Gamma, e_1 : E_1...e_n : E_n; \Delta, c : A}{c, e_1...e_n \leftarrow \text{cspawn } M\ d_1...d_n; P \vdash \Gamma, e_1 : E_1...e_n : E_n; \Delta, \Delta', c : A} \tag{CSpawn}$$

The main difference being once again that the processes here encapsulated must now also account for the unrestricted channels $e_1...e_n$, typed by $C_1...C_n$. As these channels are non-linear we don't need to capture them in a new context region.

As this language has it's genesis in the language developed by Geraldo in his work [14] all of the constructs belonging to the process layer will be typed following the rules defined in the orginal language. We only extend the language with a new typing ruleset for the constructs that we introduce, either by translation of already existing constructs into their CLL form, or by simply introducing new functionality. One major difference between both paradigms of Linear Logic, is that while in Intuitionistic Linear Logic (ILL) we can consider operations as having a *left* and *right* side, in CLL operations will have a type, and the "other side" will by typed by the corresponding*dual* operation.

The typing rules for the classical processes follow closely what is already described in the works of P. Rocha and L. Caires in [22] and here described on Chapter 2, Section 2.4.2. Below we'll present them as they are implemented in our language.

To model the exchange of information and communication between processes, we utilize the *send* and *receive* operations, typed as follows:

$$\frac{P_1 \vdash \Psi_1, y : A; \Delta; \Gamma \quad P_2 \vdash \Psi_2, x : B; \Gamma}{\mathsf{send}\, x(y.P_2); P_1 \vdash \Psi_1, \Psi_2, x : A \otimes B; \Delta; \Gamma} \quad \text{(CSend)}$$

$$\frac{Q \vdash \Psi, z : \overline{A}, x : \overline{B}; \Gamma}{\mathsf{recv}\, x(z); Q \vdash \Psi, x : \overline{A} \parr \overline{B}; \Gamma} \quad \text{(CRecv)}$$

Channel communication behaves similarly to value communication, the difference being that the type of the communicated "value" is the type of the channel being communicated (*a session type*). The typing rules follow:

$$\frac{P_1 \vdash \Delta_1, y : T_1; \Gamma \quad P_2 \vdash \Delta_2, x : T_2; \Gamma}{\mathsf{sendc}\, x(y.P_2); P_1 \vdash \Delta_1, \Delta_2, x : T_1 \otimes T_2; \Gamma} \quad \text{(CSendChan)}$$

$$\frac{Q \vdash \Delta, z : \overline{T_1}, x : \overline{T_2}; \Gamma}{\mathsf{recvc}\, x(z); Q \vdash \Delta, x : \overline{T_1} \parr \overline{T_2}; \Gamma} \quad \text{(CRecvChan)}$$

To signal termination of a process we make use of the *close* and *wait* constructs, and to signal process inaction we utilize *terminate*, all typed by the following rules:

$$\frac{}{\overline{0 \vdash \emptyset; \Gamma}} \quad \text{(Terminate)}$$

$$\frac{}{\overline{\mathsf{close}\, x \vdash x : 1; \Gamma}} \quad \text{(CClose)}$$

$$\frac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\, x; Q \vdash \Delta, x : \bot; \Gamma} \quad \text{(CWait)}$$

Branching is modeled through the *choice* and *label* constructs, that allow a choice between multiple options (a menu) to be offered and chosen from, typing follows the rules:

$$\frac{P_i \vdash \Delta, x : A_i; \Gamma}{\mathsf{case}\, x\{\, l_i.P_i \,\} \vdash \Delta, x : \&\{l_1 : A_1 ... l_i : A_i\}; \Gamma} \quad \text{(CChoice)}$$

$$\frac{P \vdash \Delta, x : A_k; \Gamma}{select\ x.l_k; P \vdash \Delta, x : \oplus\{l_1 : A_1 ... l_i : A_i\}; \Gamma} \quad \text{(CLabel)}$$

The ability to *forward* the communications from one process to another exponentially increases the expressibility of our language, and also allows us to define recursive processes, by *spawning* the process inside itself and then forwarding the communications to the newly available communication channel. This was already done in *Geraldo's* language, we just extend this functionallity to some of our new constructs. The typing rule then becomes:

$$\frac{}{\mathsf{fwd}\, x\, y \vdash x : A, y : \overline{A}; \Gamma} \quad \text{(CFwd)}$$

Unfortunately, this rule does not allow the linking of processes that model *shared memory cells* or *affine states* with dependencies. Handling such cases would require a more general rule capable of consuming both these dependencies and the channels offered by the memory cells. We leave the development of such a rule for future work.

Another very important operation that we can model with our processes is the interactions between a *server* providing a service and its *clients* consuming or interacting with it. This is done by making use of the *exponentials*, the duality inherently present in CLL allowing us to use $?A$ as a primitive, and the unrestricted context Γ. The following rules type this behaviour:

$$\frac{P \vdash \Delta, x : A; \Gamma}{!x(P) \vdash \Delta, x : !A; \Gamma} \quad \text{(Accept)}$$

$$\frac{P \vdash \Delta, x : A; \Gamma}{?x(P) \vdash \Delta, x : ?A; \Gamma} \quad \text{(Call)}$$

The ability to chose whether or not to make use of a resource, in this case a channel, is important when later we try to model shared memory. To model this functionality we allow the promotion of channels to an *affine session* which can either be used linearly or simply discarded. This promotion follows the structure of a standard promotion rule, but as some session may depend on others, we restrict the promotion to channels that either have no dependencies or that only depend on other discardable resources. This is needed in case we decide to discard the promoted resource, meaning we will neither use it nor the channels that depend on it. The typing rules that govern this behaviour are:

$$\frac{P \vdash a : A, b : \vee B; \Gamma}{\mathsf{affine}_b\, a; P \vdash a : \wedge A, b : \vee B; \Gamma} \quad \text{(Affine)}$$

$$\frac{P \vdash_\eta \Delta,\ a : A; \Gamma}{\mathsf{use}\, a;\ P \vdash \Delta,\ a : \vee A; \Gamma} \quad \text{(Use)}$$

$$\frac{}{\mathsf{discard}\, a \vdash a : \vee A; \Gamma} \quad \text{(Discard)}$$

To support operations on *affine* resources, it is necessary to introduce their dual, *coaffine* resources, thereby enabling the type system to manage both usage and disposal of linear channels consistently.

Finally we introduce the typing rules that allow us to model shared memory:

$$\frac{P \vdash \Delta, a : \wedge A; \Gamma}{\mathsf{cell}\, c(a.P) \vdash_\eta \Delta, c : S_\bullet A; \Gamma} \quad \text{(Cell)} \qquad \frac{}{\mathsf{release}\, c \vdash c : U_\bullet A; \Gamma} \quad \text{(Release)}$$

$$\frac{}{\mathsf{empty}\, c \vdash c : S_\circ A; \Gamma} \quad \text{(Empty)} \qquad \frac{Q \vdash \Delta, a : \vee A, c : U_\circ A; \Gamma}{\mathsf{take}\, c(a); Q \vdash_\eta \Delta, c : U_\bullet A; \Gamma} \quad \text{(Take)}$$

$$\frac{Q_1 \vdash \Delta_1, a : \wedge \overline{A}; \Gamma \quad Q_2 \vdash_\eta \Delta_2, c : U_\bullet A; \Gamma}{\mathsf{put}\, c(a.Q_1); Q_2 \vdash_\eta \Delta_1, \Delta_2, c : U_\circ A; \Gamma} \quad \text{(Put)}$$

Our implementation of shared state differs slightly from what is described in the work of L. Caires and P. Rocha [22]. The typing rules used to describe the system are kept the same, and the logic behind the cells themselves is similar, the only difference being there is no explicit (written) flip between a full cell and an empty cell, instead the *Cell* and *Empty* rules express creation of the cell to be interacted with, and the state (empty or full) of the cell itself is tracked and enforced during typechecking. This means we can either begin with a cell that has no content, typed by *Empty* (forcing some process to operate with a *Put* operation on that cell before any other process can interact with it) or with a full cell, typed by *Cell*, which instantly permits any process that wishes to interact with the cell to immediately do so. Every *Take* operation will reserve access to the cell, preventing other processes from interacting with the cell, and proceed to read the contents of the cell into the new channel a, thus placing the cell in the empty state. As such in order for the lock to be released and other processes to be able to interact with the cell again we must eventually execute a *Put* operation, replacing the stored, forcibly affine resource, with another one of the same type. When a process is finished interacting with the cell, it must release (*Release*) its cell usage, which must be in the unlocked state ($U_\bullet$). As such an interaction between any one process and a process modelling a *mutex shared memory cell* can be described by a *Take* operation (locking the cell), use of the stored resource, a *Put* operation (releasing the lock) and eventually a *Release* operation, dropping the process' cell usage permission.

In well-typed programs all processes interacting with the cell eventually relent access to it through *Release*, when this becomes the case the cell (and its contents, which must exist and be affine) are safely discarded.

Since cells are linear object with linear payloads in order to allow the cell to fullfil the *shared* part of it's name, we introduce a construct that allows multiple processes to interact with the cell. This construct is called *Share* and is typed by the following rules:

$$\frac{P \vdash \Delta', c : U_\bullet A; \Gamma \quad Q \vdash \Delta, c : U_\bullet A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash \Delta', \Delta, c : U_\bullet A; \Gamma} \quad \text{(Share)}$$

$$\frac{P \vdash \Delta', c : U_{\circ} A; \Gamma \quad Q \vdash \Delta, c : U_{\bullet} A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash \Delta', \Delta, c : U_{\circ} A; \Gamma} \quad \text{(ShareL)}$$

$$\frac{P \vdash \Delta', c : U_{\bullet} A; \Gamma \quad Q \vdash \Delta, c : c : U_{\circ} A; \Gamma}{\mathsf{share}\, c\{P||Q\} \vdash \Delta', \Delta, c : U_{\circ} A; \Gamma} \quad \text{(ShareR)}$$

Notice that the rules *ShareL* and *ShareR* describe a situation where at least one process is utilizing the cell, this is safe due to the locking and unlocking behaviour of the *Take* and *Put* operations.

3.3 Examples

Next we present some examples of programs written in our language. First we start with a simple example that shows how the language works.

```
1
2    c_stype List rec x. +{nil: @, cons: int ^ x};
3    c_stype Stack rec x. &{push: int => x,
4    pop: +{none: unit ^ x, some: int ^ x},
5    dispose: @};
6
7  c_decl{
8
9  null: unit -> {List}
10 fun n ->
11         c <-C {
12                 c.nil;
13                 close c
14         }
15 end
16 };
17
18 c_decl{
19 cons: int -> {List <- l:dual(List)}
20 fun a ->
21         nl <-C {
22                 nl.cons;
23                 send nl a;
24                 fwd nl l
25         }
26 end
27 };
28
```

```
29 c_decl{
30 dispose: unit -> {@ <- l:dual(List)}
31 fun u ->
32         c <-C {
33                 case l of
34                 nil: (
35                         wait l;
36                         close c
37                 )
38                 cons: (
39                         v:int <- recv l;
40                         c_ <- cspawn(dispose()) <- l;
41                         fwd c c_
42                 )
43         }
44 end
45 };
46
47 c_decl{
48 stack: unit -> {Stack <- l:dual(List)}
49 fun n ->
50         s <-C {
51                 //receive operation
52                 case s of
53                 push: (
54                         v:int <- recv s;
55                         print v;
56                         //expand list with new value node
57                         nl <- cspawn(cons v) <- l;
58                         //recursion
59                         ns <- cspawn(stack()) <- nl;
60                         fwd s ns
61                 )
62                 pop: (
63                         //Check stack length
64                         case l of
65                         nil: (
66                                 //No value, wait for list close
67                                 wait l;
68                                 //output correctly
69                                 s.none;
```

```
70                                  send s ();
71                                  //spawn null node to signify empty stack
72                                  nl <- cspawn(null());
73                                  //recursion
74                                  ns <- cspawn(stack()) <- nl;
75                                  fwd s ns
76                          )
77                          cons: (
78                                  //receive popped value
79                                  v:int <- recv l;
80                                  s.some;
81                                  send s v;
82                                  ns <- cspawn(stack()) <- l;
83                                  fwd s ns
84                          )
85                  )
86                  dispose:(
87                          ns <- cspawn(dispose()) <- l;
88                          fwd s ns
89                  )
90          }
91  end
92  };
93
94  C;;
95
96  m <-C {
97          c <- cspawn(null());
98          s <- cspawn(stack()) <- c;
99          s.push;
100         send s 1;
101         s.push;
102         send s 2;
103         s.pop;
104         case s of
105         none: (
106                 u:unit <- recv s;
107                 s.dispose;
108                 wait s;
109                 close m
110         )
```

```
111         some: (
112                 v:int <- recv s;
113                 print v;
114                 s.dispose;
115                 wait s;
116                 close m
117         )
118 }C;;
```

The program show above implements a fully concurrent, session-typed stack over a channel-based linked list represented as a recursive linear session. Each stack operation (push, pop, or dispose) is modeled as a message-passing session-typed process, with every modification spawning a new process that represents the updated stack state. The underlying list is recursively defined over channels, enabling safe, linear usage of resources and automatic disposal of closed channels. Notably, the design guarantees *LIFO* semantics while maintaining complete concurrency: multiple stack operations can coexist without shared memory, relying solely on session-typed communication to enforce correctness. This approach demonstrates how classical data structures can be reimagined in a process-oriented, type-safe setting, balancing both functional abstraction and rigorous resource management.

```
1  c_stype TSendRecv &{add:int => int => int ^ @, neg: int => int ^ @};
2
3  c_decl {
4
5  server: unit -> {!TSendRecv}
6  fun () ->
7      s <-C {
8          z <- accept s;
9          case z of
10         add: (
11                 x1:int <- recv z;
12                 x2:int <- recv z;
13                 send z (x1 + x2);
14                 close z
15         )
16         neg: (
17                 x:int <- recv z;
18                 send z (-x);
19                 close z
20         )
```

```
21
22
23      }
24  end
25
26  };
27
28  c_decl {
29
30  client: unit -> {@ <- s:?.dual(TSendRecv)}
31
32  fun () ->
33       c <-C {
34          call s(x);
35          x.neg;
36          send x 2;
37          n:int <- recv x;
38          wait x;
39          print n;
40          close c
41      }
42  end
43
44  };
45
46  c_decl {
47
48  client2: unit -> {@ <- s:?.dual(TSendRecv)}
49
50  fun () ->
51       c <-C {
52          call s(x);
53          x.add;
54          send x 2;
55          send x 3;
56          n:int <- recv x;
57          wait x;
58          print n;
59          close c
60      }
61  end
```

```
62
63 };
64
65
66 C;;
67 m <-C {
68
69 s <- cspawn (server ());
70 c, s <- cspawn (client ());
71 c2, s <- cspawn (client2 ());
72 wait c;
73 wait c2;
74 close m
75
76 }
77 C;;
```

This program shows the basic capabilities of servers, and is directly translated from an example from the *CLASS* language (named "*arithmetic-server*"). We present a fully concurrent, linear session-typed server that offers a replicated arithmetic menu to multiple clients. Each client initiates a dedicated session, selects an operation (negation or addition), sends the correct inputs, and receives the result, with linear channels ensuring single-use resource discipline. The server safely handles multiple clients concurrently, demonstrating how branching, choice, and replication can be encoded in linear session types to implement type-safe, concurrent services.

```
1  c_stype IntSR int => int ^ @;
2
3  c_decl {
4
5  cell_proc : unit -> {s_full(IntSR)}
6  fun u ->
7          f <-C {
8                  cell f (p {
9                          affine[] (p);
10                         x:int <- recv p;
11                         send p 4;
12                         print x;
13                         close p
14                         })
15
```

```
16          }
17  end
18  };
19
20  C;;
21  m <-C {
22          t <- cspawn (cell_proc ());
23          share t(
24                  take t (a);
25                  use (a);
26                  send a 4;
27                  x:int <- recv a;
28                  print x;
29                  wait a;
30                  put t (a {
31                          affine[] (a);
32                          x:int <- recv a;
33                          send a 4;
34                          close a
35                          });
36                  release t;
37                  terminate
38          ||
39                  share t (
40                          take t (a);
41                          use (a);
42                          send a 2;
43                          x:int <- recv a;
44                          wait a;
45                          print x;
46                          put t (a {
47                                  affine[] (a);
48                                  x:int <- recv a;
49                                  send a 4;
50                                  close a
51                                  });
52                          release t;
53                          close m
54                  ||
55                          release t;
56                          terminate
```

```
57                  )
58          )
59  }
60  C;;
```

This program implements a concurrent shared linear session encapsulated in a cell process, enabling multiple processes to safely interact with a single linear channel. The cell contains a simple request-response session (IntSR), where each user sends an integer, receives a response, and closes the session. Access to the cell is coordinated through exclusive acquisition (*take*) and restoration (*put*), ensuring that the linear session is consumed exactly once per use while remaining available for other processes. Multiple concurrent processes operate over the shared cell, demonstrating how linear session types can be combined with controlled sharing to implement safe, concurrent access to a single resource. The program illustrates the interplay between linear discipline, concurrency, and resource management, showing that even linear channels can be safely reused across processes without violating protocol fidelity.

Below we show the same example program, but in order to also show the affine state functionality, the stored linear session is discarded in one of the processes instead:

```
1   c_stype IntSR int => int ^ @;
2
3   c_decl {
4
5   server : unit -> {s_full(IntSR)}
6   fun u ->
7           f <-C {
8                   cell f (p {
9                           affine[] (p);
10                          x:int <- recv p;
11                          send p 4;
12                          print x;
13                          close p
14                          })
15
16          }
17  end
18  };
19
20  C;;
21  m <-C {
22          t <- cspawn (server ());
```

```
23          share t(
24                  take t (a);
25                  use (a);
26                  send a 4;
27                  x:int <- recv a;
28                  print x;
29                  wait a;
30                  put t (a {
31                          affine[] (a);
32                          x:int <- recv a;
33                          send a 4;
34                          print x;
35                          close a
36                          });
37                  release t;
38                  terminate
39          ||
40                  share t (
41                          take t (a);
42                          discard (a);
43                          put t (a {
44                                  affine[] (a);
45                                  x:int <- recv a;
46                                  send a 4;
47                                  print x;
48                                  close a
49                                  });
50                          release t;
51                          close m
52                  ||
53                          release t;
54                          terminate
55                  )
56          )
57  }
58  C;;
```

4

Implementation

As this work builds directly upon the system developed by J. Geraldo in [14], we decide it would be best to adopt and extend his original framework. As such, we make use of the OCaml language to implement a *two-stage* pipeline consisting of *type checking* and *compilation*, to ultimately produce executable *Go code*.

For the typechecking stage, we preserve Geraldo's system as a foundation and extend it in two key ways: first, by translating the constructs he introduces into CLL, and second, by incorporating new constructs specific to our language, all in accordance with the typing rules presented in Section 3.2.

In the compilation stage, we first process the *Abstract Syntax Trees* produced by the typechecker, transforming the *Session Types* within them into sets of specialized, state-like structures with some associated methods in the target language. We then translate the program instructions into executable Go code that operates over these generated states.

4.1 Typechecker

The typechecker is built around the technique of bidirectional typing, that was already used by J.Geraldo. Using this technique means we can both infer and check the type of our language's constructs. This allows to express our typing judgments, expressed by $\Psi \Vdash M : T, P \vdash \Delta, c : A$ and $P \vdash \Delta, \Gamma, c : A$ through implementable algorithmic expressions for *synthesis* ($\Psi \Vdash M \Rightarrow T$) and *checking* ($\Psi \Vdash M \Leftarrow T$) of types in our system. Its worth noting that although formally our language utilizes the typing judgments expressed as $P \vdash \Psi, \Delta$ or $P \vdash \Psi, \Gamma, \Delta$. In practice we expanded on what was already implemented by Geraldo in his work, which is more accurately described by $\Psi, \Delta \vdash P :: c : A$ or $\Psi, \Delta, \Gamma \vdash P :: c : A$ respectively. Although the typechecking behaviour is similar for both forms, formally this makes a difference when describing the typing rules.

The typing rules applied to the functional layer of our language follow the standard, well-documented approach to bidirectional typing, and therefore we omit the rules pertaining to these judgments. We consider, as mentioned previously, the following algorithmic judgments for the functional layer:

- $\Psi \Vdash M \Rightarrow T$ - which states that M *synthesizes* type T, under context Φ;
- $\Psi \Vdash M \Leftarrow T$ - which states that M *checks* against type T, under context Φ;

Under this typechecking system, every rule must either be a *synthesis* or a *checking* rule. An immediate consequence of this fact is the possibility of omitting type annotations from most of the definitions and bindings in our language (top-level definitions like function definition must still be type annotated, along with some other constructs).

When trying to typecheck the process layer we must also account for the associated contexts, in particular, our algorithmic rules must now account for the linear context Δ, which must be handled linearly, and the unrestricted context Γ. To handle the linear context we follow what was already implemented in [14] and originally formulated in [11]. Their approach was to force the consumption/use of *all* linear resources present in the context through separation of said context into an *input* linear context, which must be used in the process, and an *output* linear context, which tracks the leftovers (and in some cases newly produced resources) that must be used in the continuation. For a program to be considered "well-typed" this output context must be empty by the time the program terminates, meaning all resources were fully used up in the evaluation and thus guaranteeing linearity. As to the handling of the unrestricted context Γ, resources present here can be used multiple (possibly zero) times, which means we must simply ensure that if a resource is used it must exist in the context. Because no process can ever consume the resources present in this context, the unrestricted context may be non-empty by the time the program terminates.

The algorithmic judgments must, in this layer, be adapted to reflect this context handling. When working on programs that only make use of the ILL constructs present in the language, we can utilize the rules already described in [14] ($\Psi, \Delta_I / \Delta_O \Vdash M \Rightarrow T$ and $\Psi, \Delta_I / \Delta_O \Vdash M \Leftarrow T$), which describe the context splitting behaviour mentioned previously. As to programs that utilize the new constructs, thus also making use of the new unrestricted context, we must further alter the judgments by introducing the context into the rule, as such, $\Psi, \Gamma, \Delta_I / \Delta_O \Vdash M \Rightarrow T$ and $\Psi, \Gamma \Delta_I / \Delta_O \Vdash M \Leftarrow T$ become the new typing judgments for *synthesis* and *checking* respectively.

To properly convey the implications of our changes, we now show an example of how the algorithmic translations of the typing rules for process spawning look like:

$$\frac{\Psi \vdash M \Rightarrow \{B_1 ... B_n \vdash A\} \quad \Delta_I' = \Delta_I \backslash \{d_1 ... d_n : B\} \quad \Psi; \Delta_I', c : A / \Delta_O \vdash P \Rightarrow C}{\Psi; \Delta_I / \Delta_O \vdash \{c \leftarrow \text{spawn } M\, d_1 ... d_n; P\} \Rightarrow C} \quad \text{(Spawn)}$$

This rule is comparable to what J.Geraldo had already described in his work for the same purpose of process composition, and expresses that we can synthesize the type of the $c \leftarrow \texttt{spawn}\ M\, d_1 ... d_n; P$ process by first synthesizing the type of the expression M. This then allows us to calculate the input context of P (the continuing process), Δ_I', which

by adding the channel c produced by the *spawn* process, and simultaneously removing the input channels $d_1 ... d_n$ allows us to eventually (and in practice recursively), synthesize the type of P.

The rule bellow is equivalent but for programs that must also handle an unrestricted context. It is worth noting that the context is passed along without the distinction of *input* and *output*. One of the main advantages of integrating CLL into the system is that we can now "output" a result set instead of a single channel. In this case in particular, notice that the outer *cspawn* process also outputs a (possibly empty) set of new non-linear resources, which are then added to the appropriate context, Γ, to be used by the continuation P.

$$\frac{\Psi \vdash M \Rightarrow \{B_1 ... B_n, E_1 ... E_n \vdash A\} \quad \Delta_I' = \Delta_I \backslash \{d_1 ... d_n : B\} \quad \Psi; \Gamma, e_1 ... e_n : E; \Delta_I', c : A / \Delta_O \vdash P \Rightarrow C}{\Psi; \Gamma; \Delta_I / \Delta_O \vdash \{c, e_1 ... e_n \leftarrow \text{cspawn } M\, d_1 ... d_n; P\} \Rightarrow C} \quad \text{(CSpawn)}$$

In order to properly bridge the gap between the synthesizing and checking of types during the typechecking process we make use of the following rules, adapted from the previous work:

$$\frac{\Psi; \Delta; \Gamma \vdash P \Rightarrow c : A \quad A \leq B}{\Psi; \Delta; \Gamma \vdash P \Leftarrow c : B} \quad \text{(Sub)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash P \Leftarrow c : A}{\Psi; \Delta; \Gamma \vdash (P :: c : A) \Rightarrow c : A} \quad \text{(Annot)}$$

The *Sub* rule dictates that if a process P can be synthesized as being of type A, and if A is a subtype of another type B, then we can simply check P against the more general type B. This is particularly useful during recursion where a process like $\mu X.\&\{l_1 : int^{(}\mu X.\&\{l_1 : int^X, l_2 : int^1\}), l_2 : int^1\}$ can be *folded* and reinterpreted as $\mu X.\&\{l_1 : int^X, l_2 : int^1\}$. Typechecking this kind of recursion can quickly become a problem, the solution is to treat the first process as one offering a session channel of the *folded type* of the second. This then becomes easy to typecheck with the *Sub* rule.

The *Annot* rule is simply the method we use to handle type annotations in our system. The rule dictates that if a process P can be checked against type A, then that type can safely by synthesized from P.

The concept of duality is, as explained before, how we naturally handle each end of the channels' "endpoints". As it is comparable to the linear negation of an expression we decided to, in practice, implement it like so:

```
1 let construct_dual_stype stype env =
2   match stype with
```

```
3    | STEnd -> STWait
4    | STWait -> STEnd
5    | STVar _ -> stype
6    | STExtChoice l -> STIntChoice (construct_dual_list l env)
7    | STIntChoice l -> STExtChoice (construct_dual_list l env)
8    | STSend (t, st) -> STRecv (t, construct_dual_stype st env)
9    | STRecv (t, st) -> STSend (t, construct_dual_stype st env)
10   | STSendChan (st1, st2) ->
11     STRecvChan (construct_dual_stype st1 env, construct_dual_stype st2 env)
12   | STRecvChan (st1, st2) ->
13     STSendChan (construct_dual_stype st1 env, construct_dual_stype st2 env)
14   | STRec (v, st) -> STRec (v, construct_dual_stype st env)
15   | STUVar v -> begin match List.assoc_opt v env with
16                 | Some TProc (st, _) -> construct_dual_stype st env
17                 | Some TClassProc (st, _, _) -> construct_dual_stype st env
18                 | _ -> error (NoSuchArg v)
19                end
20   | STSvrDefVar v -> STClntInvVar (construct_dual_stype v env)
21   | STClntInvVar v -> STSvrDefVar (construct_dual_stype v env)
22   | STDual v -> construct_dual_stype v env
23   | STAffine (st, deps) -> STCoAffine (construct_dual_stype st env, deps)
24   | STCoAffine (st, deps) -> STAffine (construct_dual_stype st env, deps)
25   | STFullState (st) -> STFullUse (construct_dual_stype st env)
26   | STEmptyState (st) -> STEmptyUse (construct_dual_stype st env)
27   | STFullUse (st) -> STFullState (construct_dual_stype st env)
28   | STEmptyUse (st) -> STEmptyState (construct_dual_stype st env)
29   | STMultiRecv (_,_) | STMultiSend (_,_) -> assert false
```

Every type has its dual mapped out and when given a Session Type this function will recursively dualize it. We also need to pass along the functional environment Ψ, *env*, in case we are trying to dualize a recursive session type.

Our typechecker implements bidirectional typechecking with two different functions (one for checking another for synthesizing the types of functional expressions) with the following signatures:

```
1  let rec synth lin_ctxt env e used_vars: (var * stype) list ->
2  (var * ty) list -> exp -> var list -> exp * ty * var list
3
4
5  let rec check lin_ctxt env e t used_vars: (var * stype) list ->
6  (var * ty) list -> exp -> ty -> var list -> exp * ty * var list
```

Types in our language extend the native *OCaml* types (e.g. *string* and *list*) and those introduced in Geraldo's implementation (*ty*, *exp*, *stype*, *proc*). Our additions consist mainly of a new *c_proc* type to represent process layer connectives that consist of only classical operations (defined in Section 3.1, Figure 3.2) and some expansions on the already existing types, specifically:

- *exp* and *ty* → where we define a new functional value to represent classical process functional expressions
- *stype* → where we expand the definition of this type to accommodate for our new session types

The previously defined type of *proc* no longer represents a general process of the language, and instead completes *c_proc*, in that it represents the remaining process layer connectives, those that contain only intuitionistic operations (defined in Section 3.1, Figure 3.2).

These changes to the typing system of our language are reflected in the functions used to synthesize and check terms belonging to the functional layer. Those adaptations are presented here:

```
1  let rec check lin_ctxt env e t used_vars =
2    match e with
3    | ClassProcExp (chan, proc, _, _, _) ->
4           begin match t with
5           | TClassProc (st, lin_ctxt, unr_ctxt) ->
6             let (out_lin_ctxt, _, p', st', vars) =
7               synth_class_proc unr_ctxt lin_ctxt env proc chan used_vars in
8               if out_lin_ctxt = [] then
9                 if subtyping [] st st' || subtyping [] st' st then
10                  (ClassProcExp (chan, p', Some st, lin_ctxt, unr_ctxt),
11                    t, vars)
12                else error (NonMatchingSTypes (st, st'))
13              else error (NonEmptyLinearContext)
14          | _ -> error (NotProcessType t)
15          end
16
17 let rec synth lin_ctxt env e used_vars =
18 match e with
19 | ClassProcExp (chan, proc, _, _, unr_ctxt) ->
20   let (out_lin_ctxt, _, p', st, vars) =
21     synth_class_proc unr_ctxt lin_ctxt env proc chan used_vars in
22     if out_lin_ctxt = [] then
```

```
23          (ClassProcExp (chan, p', Some st, lin_ctxt, unr_ctxt),
24            TClassProc (st, lin_ctxt, unr_ctxt), vars)
25        else error (NonEmptyLinearContext)
26    | ExecExp exp -> (
27        let e', ty, vars = synth lin_ctxt env exp used_vars in
28        match ty with
29        | TClassProc _ -> (ExecExp e', ty, vars)
30    )
```

These functions take as parameters the expression to check or synthesize, a lin_ctxt with all the linear resources, a functional environment, and a list of used variable identifiers, used only during the compilation phase. Because the same two functions are used for both classical and intuitionistic programs, along with these parameters, we also need to reference the unrestricted context Γ, which keeps track of all the non-linear channels, when typing a functional value of the classical form. This context is obtained from the expression itself. The checking process also requires the type to check against which is passed in the form of the parameter t. These *check* and *synth* functions model the typing judgments of the functional layer explained previously.

The return value of both of these functions is the functional expression itself, expanded with some additional type information, the evaluated type of the expression and the aforementioned list of used identifiers, wrapped in a tuple.

For the process layer things work a bit differently, firstly we utilize distinct functions for classical and intuitionistic processes, and secondly we synthesize expressions through the use of a *synth_proc* (or *synth_class_proc* for classical process expressions) function which models the typing judgment just as in the functional layer, but have no real function to model the judgment for checking. Instead, like in Geraldo's work we utilize a combination of the synthesizing functions and the previously mentioned subtyping function to compare session types. This happens because of the possibility for session types to be recursive. For the sake of completeness we will explain the checking process here as well, while also introducing the slight variation our typing discipline requires.

```
1   let rec check lin_ctxt env e t used_vars =
2   match e with
3   | _ ->
4       let e', t', vars = synth lin_ctxt env e used_vars in
5       if ty_eq t' t then (e', t', vars) else error (UnexpectedType (t', t))
```

We start the checking process by calling on the *check* function, which pattern matches our process expression to the *wildcard pattern*. Then the expression is synthesized using the functional layer *synth* function, which returns a type t'. This type is in turn checked against the type t, by calling the function *ty_eq* returning a tuple with the expression

(expanded with types), the previously synthesized type, and the list of used variables, or an error.

```
let rec ty_eq t1 t2 =
match (t1, t2) with
| TProc (p1, ctxt1), TProc (p2, ctxt2) ->
    match_lin_ctxt ctxt1 ctxt2 &&
      (subtyping [] p1 p2 || subtyping [] p2 p1)
| TClassProc (p1, lin_ctxt_1, unr_ctxt_1),
  TClassProc (p2, lin_ctxt_2, unr_ctxt_2) ->
    (match_unr_ctxt unr_ctxt_1 unr_ctxt_2 ||
      match_unr_ctxt unr_ctxt_2 unr_ctxt_1) &&
    (subtyping [] p1 p2 || subtyping [] p2 p1) &&
      match_lin_ctxt lin_ctxt_1 lin_ctxt_2
| _ -> t1 = t2
```

The function *ty_eq* starts by matching the two types *t1* and *t2* to either *TProc*, the type of intuitionistic process expressions, or *TClassProc*, the type of classical process expressions. Since we are typing two process expressions (classical or not), one of these patterns must match. Note that *p1* and *p2* represent the session types of the processes. Next the function will evaluate if the two processes are considered equal by:

- If both *t1* and *t2* are of type *TProc*:
  - Checking that *p1* is a subtype of *p2*, or vice-versa
  - Checking that *ctxt1* and *ctxt2*, the linear contexts of the processes, are the same, meaning they have the same number of elements and that for every pair (channel, session type) present in *ctxt1* the same pair is present in *ctxt2*, or vice-versa
- If both *t1* and *t2* are of type *TClassProc*:
  - Checking that *p1* is a subtype of *p2*, or vice-versa
  - Checking that *unr_ctxt_1* is a subset of *unr_ctxt_2* (or vice-versa), meaning it is sufficient that for all pairs (channel, session type) that are present in *unr_ctxt_1* they are also contained in *unr_ctxt_2* (or vice-versa).
  - Checking that *lin_ctxt1* and *lin_ctxt2*, the linear contexts of the processes, are the same, meaning they have the same number of elements and that for every pair (channel, session type) present in *ctxt1* the same pair is present in *ctxt2*, or vice-versa

The signatures for the remaining functions are:

```
let rec subtyping ctxt st1 st2 =
if List.mem (st1, st2) ctxt then true
else
  match (st1, st2) with
  | STEnd, STEnd -> true
  | STWait, STWait -> true
  | STRec (x, t), u ->
      subtyping ([ (STRec (x, t), u) ] @ ctxt) (unfold (STRec (x, t))) u
  | t, STRec (x, u) ->
      subtyping ([ (t, STRec (x, u)) ] @ ctxt) t (unfold (STRec (x, u)))
  | STRecv (tl, t), STRecv (ul, u) -> ty_eq tl ul && subtyping ctxt t u
  | STSend (tl, t), STSend (ul, u) -> ty_eq ul tl && subtyping ctxt t u
  | STExtChoice ls, STExtChoice lt ->
      let map_fun (l, s) =
        begin match List.assoc_opt l lt with
        | Some t -> subtyping ctxt s t
        | None -> false end
      in
      List.for_all map_fun ls
  | STIntChoice ls, STIntChoice lt ->
      let map_fun (l, t) =
        begin match List.assoc_opt l ls with
        | Some s -> subtyping ctxt s t
        | None -> false end
      in
      List.for_all map_fun lt
  | STRecvChan (tl, t), STRecvChan (ul, u) ->
    subtyping ctxt tl ul && subtyping ctxt t u
  | STSendChan (tl, t), STSendChan (ul, u) ->
    subtyping ctxt ul tl && subtyping ctxt t u
  | STMultiRecv (tl, t), STMultiRecv (ul, u)
  | STMultiSend (tl, t), STMultiSend (ul, u)
    ->
      multi_eq tl ul && subtyping ctxt t u
  | STSvrDefVar t1, STSvrDefVar t2 ->
    subtyping ctxt t1 t2
  | STClntInvVar t1, STClntInvVar t2 -> subtyping ctxt t1 t2
  | STDual t1, STDual t2 -> subtyping ctxt t1 t2
  | STAffine (st1, _), STAffine (st2, _) -> subtyping ctxt st1 st2
  | STCoAffine (st1, _), STCoAffine (st2, _) -> subtyping ctxt st1 st2
  | STFullState (st1), STFullState(st2) -> subtyping ctxt st1 st2
```

```
42      | STEmptyState (st1), STEmptyState(st2) -> subtyping ctxt st1 st2
43      | STFullUse (st1), STFullUse(st2) -> subtyping ctxt st1 st2
44      | STEmptyUse (st1), STEmptyUse(st2) -> subtyping ctxt st1 st2
45      | _ -> false
```

This function evaluates if a session type *st1* in a subtype of another session type *s2*, modelling the *Sub* judgment described earlier.

```
1   let rec match_lin_ctxt ctxt1 ctxt2 =
2   if List.length ctxt1 = List.length ctxt2 then
3     let matching_function (c, st) =
4       match List.assoc_opt c ctxt2 with
5       | Some st' -> subtyping [] st st' || subtyping [] st' st
6       | None -> false
7     in
8     List.for_all matching_function ctxt1
9   else false
10
11  let rec match_unr_ctxt ctxt1 ctxt2 =
12  List.for_all (fun unr_arg ->
13      List.exists (fun ctxt_chan ->
14        unr_arg == ctxt_chan) ctxt2) ctxt1
```

These functions check the contexts, linear and unrestricted respectively, in order to ascertain if they can be considered equal, in order to ensure the behaviour described previously.

For the synthesis side of the process layer expressions we make use of two functions, one is the already previously implemented *synth_proc* which is responsible for synthesizing the type of intuitionistic process expressions, referred to as *ProcExp*. The other one is the new *synth_class_proc* which is responsible for synthesizing classical process expressions, in our implementation called *ClassProcExp*. Below are the signatures for both functions:

```
1   let rec synth_proc lin_ctxt env proc c used_vars :
2     (var * stype) list -> (var * ty) list ->
3     proc -> var -> var list ->
4     (var * stype) list * proc * stype * var list
5
6   let rec synth_class_proc unr_ctxt lin_ctxt env proc c used_vars :
7     (var * stype) list ->
8     (var * stype) list -> (var * ty) list -> c_proc -> var -> var list ->
9     (var * stype) list * (var * stype) list * c_proc * stype * var list
```

It is important to note that these functions are the implementations of the typing judgments described and explained in Subsection 3.2.2, where each judgment will correspond to a different branch in the synthesizing function's *match-block*.

Because the behaviour of the synthesis function for intuitionistic processes remains unchanged from that described in [14], and its operation closely parallels that of the classical process synthesis function introduced here, our discussion primarily focuses on the *synth_class_proc* function. When relevant, we provide observations and comparisons with the behaviour described in [14].

We must pass as parameters to the synthesis function the unrestricted context (only for *synth_class_proc*), linear context and functional environment under which we will synthesize the type of the process, along with the process expression itself, the name of the session offered by the process, and finally the list of used identifiers to be used during compilation. The function returns a tuple consisting of:

- The unrestricted context with all the non-linear sessions used during the process (if the expression is a classical process)
- The output linear context, which as mentioned before will contain the channels that will be used in the continuation
- The process expression, which is expanded with some type annotations
- The process' offered session's type
- The list of used identifiers, to use during compilation

For clarity we now detail a couple of relevant examples of said process:

```
1  synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
2  match proc with
3  | CSend (c', e, _, p) ->
4      if c = c' then
5        let (out_lin_ctxt, out_unr_ctxt, p', st, p_vars) =
6          synth_class_proc unr_ctxt lin_ctxt env p c used_vars in
7          let (e', t, e_vars) = synth [] env e used_vars in
8            (out_lin_ctxt, out_unr_ctxt, CSend (c', e', Some t, p'),
9              STSend (t, st), p_vars@e_vars)
10     else
11       begin match List.assoc_opt c' lin_ctxt with
12       | Some folded ->
13           begin match unfold folded with
14           | STSend (t, st) ->
15             begin match check [] env e t used_vars with
16             | (e', _, e_vars) -> let new_lin_ctxt =
```

```
17                (c', st)::(List.remove_assoc c' lin_ctxt) in
18                let (out_lin_ctxt, out_unr_ctxt, p', st', p_vars) =
19                  synth_class_proc unr_ctxt new_lin_ctxt env p c used_vars in
20                  (out_lin_ctxt, out_unr_ctxt, CSend (c', e', Some t, p'),
21                    st', p_vars@e_vars)
22              end
23            | wrong -> error (ChannelHasWrongSType (c', wrong))
24            end
25        | None -> error (NoSuchChannelInContext c')
26        end
```

In this example we show what happens when the expression matches a process that sends a value *CSend* (line 3). First we check if the value will be sent through a channel that is offered by the process currently being type, or if it is sending it through a channel present in the linear context. Although in CLL there is no such distinction (the channel simply exists in the context, and is then used), here we decided to expand upon the system that was already designed for ILL, and as such this definition of channel ownership bleeds through into our implementation.

If the channel is owned by the process we are typing, we synthesize the continuation process p, producing the tuple described previously. We then consider the expression e, representing the value being sent, and synthesize its type (line 6). This synthesis yields a tuple containing the annotated expression e', the type of the expression t, and a list of used variables, where e' is identical to e but may include additional type annotations.

Once these components have been obtained, the initial process can be typed. The algorithm returns a tuple with five elements: $(\Delta_O, \Gamma_O, \mathtt{p'}, S, U)$. Here Δ_O denotes the resulting linear context, while Γ_O denotes the resulting unrestricted context, which contains at least the channels present in the original unrestricted context and possibly additional ones. The element p' represents the expanded process expression, which incorporates the annotated expression e' with type t as well as the expanded continuation process expression. The element S corresponds to the session type offered by the process, which in this case is *STSend* (which represents the type $T \wedge A$ mentioned in Section 3.2.1) extended with the type of the value being sent and the session type offered by the continuation process. Finally, U denotes the list of variables used during typing, obtained by concatenating the lists produced in the previous synthesis steps.

If a channel instead appears exclusively in the *input* linear context, this indicates that the session is being offered by another process. In this scenario, we must retrieve the channel from the context, unfold it if it is a recursive session type, and match the session type with the appropriate behaviour. Since we are sending a value, the expected behaviour is *STSend*. The unfolding of the recursive type is done through the use of the *unfold* function, which if a session type is recursive returns the inner, non-recursive session type, and if it is not recursive does nothing.

For the intuitionistic variant, where there is a clear distinction between *left* and *right* rules for *receiving* or *offering* a session, the type to check against is actually *STRecv* (as shown below), which represents the implementation of type $T \supset A$. This is because, when the session is offered by another process, that process is expected to receive a value from the current process (which has been matched to a *Send* operation).

In contrast, CLL does not reflect this left/right distinction, which corresponds to channel ownership. Therefore, it is sufficient to verify that the session being used represents a *send* operation; the process on the other end of the channel will naturally perform the dual operation, *recv*. This exemplifies one of the practical advantages of using CLL.

During the typing process the function can return errors if either the channel is not offered by the process and is not present in the input linear context, or if the channel with name *c'* is present in the linear context, but does not have the correct session type associated.

```
1  synth_proc lin_ctxt env proc c used_vars =
2    match proc with
3    | Send (c', e, _, p) -> (
4      if
5          c' = c then
6        let out_ctxt, p', st, p_vars = synth_proc lin_ctxt env p c used_vars in
7        let e', t, e_vars = synth [] env e used_vars in
8        (out_ctxt, Send (c', e', Some t, p'), STSend (t, st), p_vars @ e_vars)
9      else
10       match List.assoc_opt c' lin_ctxt with
11       | Some folded -> (
12         match unfold folded with
13         | STRecv (t, st) -> (
14           match check [] env e t used_vars with
15           | e', _, e_vars ->
16             let new_lin_ctxt = (c', st) :: List.remove_assoc c' lin_ctxt in
17               let out_ctxt, p', st', p_vars =
18                 synth_proc new_lin_ctxt env p c used_vars
19               in
20             (out_ctxt, Send (c', e', Some t, p'), st', p_vars @ e_vars))
21         | wrong -> error (ChannelHasWrongSType (c', wrong)))
22       | None -> error (NoSuchChannelInContext c'))
```

For completeness we show what happens when the process matches the dual operation *CRecv*. Notice that in this case an annotation of type must be present in the process' expression for typing to be possible, meaning we must specify with an annotation the type of the functional value to be received, otherwise an error is thrown. For compilation

purposes, we can also observe that if the value that is received is then never used by another process we return a special process expression is returned in the tuple.

```
synth_proc lin_ctxt env proc c used_vars =
  match proc with
  | CRecv (v, c', Some t,  p) ->
      if c' = c then
        let (out_lin_ctxt, out_unr_ctxt, p', st, vars) =
        synth_class_proc unr_ctxt lin_ctxt ((v, t)::env) p c used_vars in
          begin match List.find_opt (fun a -> a = v) vars with
          | None ->
            (out_lin_ctxt, out_unr_ctxt,
              CRecv ("_", c', Some t, p'), STRecv (t, st), vars)
          | Some _ ->
            (out_lin_ctxt, out_unr_ctxt,
              CRecv (v, c', Some t, p'), STRecv (t, st), vars)
          end
      else
        begin match List.assoc_opt c' lin_ctxt with
        | Some folded ->
          begin match unfold folded with
          | STRecv (t', st) ->
            if
              ty_eq t t'
            then
              let new_lin_ctxt =
                (c', st)::(List.remove_assoc c' lin_ctxt) in
                let (out_lin_ctxt, out_unr_ctxt, p', st', vars) =
                  synth_class_proc unr_ctxt new_lin_ctxt
                    ((v, t)::env) p c used_vars in
                  begin match List.find_opt (fun a -> a = v) vars with
                  | None ->
                    (out_lin_ctxt, out_unr_ctxt,
                      CRecv ("_", c', Some t, p'), st', vars)
                  | Some _ ->
                    (out_lin_ctxt, out_unr_ctxt,
                      CRecv (v, c', Some t, p'), st', vars)
                  end
            else
              error (UnexpectedType (t, t'))
          | wrong -> error (ChannelHasWrongSType (c', wrong))
```

```
39            end
40          | None -> error (NoSuchChannelInContext c')
41          end
42    | CRecv (_, _, None, _) -> error (CannotInferType)
```

When trying to close a session the process will match to either a *CClose* or a *CWait* process, depending on which side of the session endpoint we're typing. If the process is matched to *CClose*, then we are trying to shut down communication through the channel. As such we must own the channel, and the typing process will simply return the linear context, the unrestricted context, the process itself (with no new annotations), the type of the process *STEnd* (equivalent to the type *1* in the formal syntax) and a list of used identifiers. If we try to close a channel that is not owned by the process, then an error is thrown.

```
1  synth_proc lin_ctxt env proc c used_vars =
2    match proc with
3    | CClose c' ->
4        if c' = c then (lin_ctxt, unr_ctxt, proc, STEnd, used_vars)
5        else error (CannotCloseUnownedChannel c')
6    | CWait (c', p) ->
7      if
8        c' = c
9      then
10       error (CannotWaitOwnChannel c')
11     else
12       begin match List.assoc_opt c' lin_ctxt with
13       | Some folded ->
14         begin match unfold folded with
15         | STWait -> let new_lin_ctxt = List.remove_assoc c' lin_ctxt in
16           let (out_lin_ctxt, out_unr_ctxt, p', st', p_vars) =
17             synth_class_proc unr_ctxt new_lin_ctxt env p c used_vars in
18             (out_lin_ctxt, out_unr_ctxt, CWait (c', p'), st', p_vars)
19         | wrong -> error (CannotWaitChannelOfType (c', wrong))
20         end
21       | None -> error (NoSuchChannelInContext c')
22       end
```

When on the other side of the session's "endpoint", if a process matches to *CWait* it means that the process will no longer interact through that channel and is waiting for the channel to close. As such the channel cannot belong to the process itself, and must then be present in the linear context with the correct *STWait* type (which represents the $\perp$ type from the syntax), or an error is thrown. When the channel is found in the linear context

with the correct session type, we proceed to remove it from the linear context (it is now shut down) and then continue with the synthesis of the continuation process *p*. After that synthesis we return a tuple with the output linear context (without the closed channel), the output unrestricted context, the expanded process expression, the synthesized session type, and the list of used variables to be used during compilation.

One of the most important constructs of the language is the *spawn* and *cspawn* constructs, which we have extensively talked about before in Section 3.1.3 and the beginning of this Chapter. There are some major differences between the intuitionistic *Spawn* process and the *CSpawn* process, due to the fact we must handle the passing of channels from the linear to the unrestricted context (simulating the unrestricted cut rule as mentioned before).

```
1  synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
2    match proc with
3    | CSpawn (c', e, _, p, unr_args, args) ->
4      if
5        c = c'
6      then
7        error (ChannelAlreadyExists c')
8      else
9        begin match List.assoc_opt c' lin_ctxt with
10       | Some _ -> error (ChannelAlreadyExists c')
11       | None ->
12         begin match List.assoc_opt c' unr_ctxt with
13         | Some _ -> error (ChannelAlreadyExists c')
14         | None -> let (spawn_exp_ctxt, next_lin_ctxt) =
15           split_ctxt_for_spawn args lin_ctxt in
16             let (e', t, e_vars) = synth spawn_exp_ctxt env e used_vars in
17               let (new_unr_ctxt, next_lin_ctxt) =
18                 if
19                   check_spawn_unr_args unr_args unr_ctxt
20                 then
21                   (unr_ctxt, next_lin_ctxt)
22                 else
23                   update_unr_ctxt unr_args unr_ctxt next_lin_ctxt
24                 in
25                   begin match t with
26                   | TClassProc (st, expected_lin_ctxt, _) ->
27                     if (compare_lin_ctxts expected_lin_ctxt
28                         (spawn_exp_ctxt @ new_unr_ctxt))
29                     then
30                     (
```

```
31                      let new_chan_stype = (construct_dual_stype st env) in
32                        let new_lin_ctxt =
33                          (c', new_chan_stype)::next_lin_ctxt in
34                          let (out_lin_ctxt, out_unr_ctxt, p', st', p_vars) =
35                            synth_class_proc new_unr_ctxt new_lin_ctxt env p c
36                              used_vars in
37                            (out_lin_ctxt, out_unr_ctxt,
38                              (CSpawn (c', e', Some new_chan_stype, p',
39                                unr_args, args)),
40                              st', e_vars@p_vars)
41                    )
42                    else error (
43                      LinearContextsMismatched (
44                        expected_lin_ctxt, spawn_exp_ctxt @ new_unr_ctxt)
45                    )
46                  | ty -> error (NotProcessType ty)
47                  end
48          end
49        end
```

Above is the procedure to type a classical process that spawns other processes. First we must check that the channel the newly spawned process will offer does not already exist or an error is thrown. After that we start by splitting the input linear context into two disjoint contexts: the *spawn_exp_ctxt*, which represents the linear resources to be used inside the spawned process' expression, and the *next_lin_ctxt* which represents the linear resources that are leftover, and will be used by the continuation p of the process currently being typed. This split is done through the use of the *split_ctxt_for_spawn* function, which takes the list of channel names *args* and the input linear context, and proceeds to split all those channels into the new *spawn_exp_ctxt*.

After splitting the linear contexts, we proceed to synthesize the type of the functional expression e that will spawn the new process. This step will return a tuple containing the expanded expression e', the type of that expression t, and a list of variable names used in the expression. Then we check to see if all the non-linear channels that will be used by the spawned process are already contained in the unrestricted context. If they are not we will need to grab them from the linear context (where they must be present or an error is thrown) and place them in the unrestricted context, this is done through the use of the function *update_unr_ctxt* which is shown below. This function returns both the linear context and the unrestricted context with the proper channels removed or added.

```
1  update_unr_ctxt vars unr_ctxt lin_ctxt =
2    List.fold_left(fun (acc_unr, acc_lin) chan_name ->
```

```
3       begin match List.assoc_opt chan_name acc_unr with
4       | Some _ -> (acc_unr, acc_lin)
5       | None ->
6         begin match List.assoc_opt chan_name lin_ctxt with
7         | Some st -> let new_unr_ctxt = (chan_name, st) :: acc_unr in
8           let new_lin_ctxt = List.remove_assoc chan_name acc_lin in
9             (new_unr_ctxt, new_lin_ctxt)
10        | None -> error (NoSuchChannelInContext chan_name)
11        end
12      end) (unr_ctxt, lin_ctxt) vars
```

After the update of the contexts or if all the non-linear channels were already present in the correct context, we check that the expression is a classical process, by matching its type t with *TClassProc*. We then confirm that the context resulting from the union of the unrestricted context with the split *spawn_exp_ctxt* matches the linear context expected by the spawned process exactly, taking into account the order in which channels appear. Next we need to place the session offered by the spawned process into the linear context, with the correct session type, which will be the dual of st. This channel must be dualized before being added to the linear context, because the spawned process and the continuation process will interact with each other, and as such the continuation process p will execute the dual operations of the spawned process, thus needing the new session c' to be typed as the dual of the session type of the spawned process, st.

Finally after adding the channel to the linear context, we can synthesize the type of the continuation p, and return the resulting tuple with the output linear context, the output unrestricted context, the expanded process expression, the type of the offered session, and the list of variables used in both processes.

Because in the intuitionistic version we do not have to worry about the non-linear resources, the procedure is much simpler. We check that the new session doesn't just like before, and if it does not we proceed to split the input linear context in much the same way. After that we also synthesize the type of the functional expression which will originate the process, but as opposed to the classical version, we immediately check that it corresponds to an intuitionistic process *TProc*, if that is the case we directly add the channel to the linear context and proceed with the synthesis of the type of p, the continuation process, returning a tuple with the output linear context, the expanded process expression, the offered session type, and the list of used variables.

```
1   synth_proc lin_ctxt env proc c used_vars =
2     match proc with
3     | Spawn (c', e, _, p, args) -> (
4       if
5         c' = c
```

```
6       then
7         error (ChannelAlreadyExists c')
8       else
9         match List.assoc_opt c' lin_ctxt with
10        | Some _ -> error (ChannelAlreadyExists c')
11        | None -> (
12          let spawn_exp_ctxt, next_lin_ctxt =
13            split_ctxt_for_spawn args lin_ctxt in
14          let e', t, e_vars = synth spawn_exp_ctxt env e used_vars in
15          match t with
16          | TProc (st, _) ->
17            let new_lin_ctxt = (c', st) :: next_lin_ctxt in
18            let out_ctxt, p', st', p_vars =
19              synth_proc new_lin_ctxt env p c used_vars
20            in
21              (out_ctxt, Spawn (c', e', Some st, p', args), st',
22                e_vars @ p_vars)
23          | ty -> error (NotProcessType ty)))
```

When a process matches to *CChoice*, we are offering a choice to some other process, which must in turn send a label (*CLabel*) in order to proceed. While in the intuitionistic version of this process there is a distinction between internal and external choice, in the classical version of this process, no such distinction exists. Instead the logic is that one process selects an option (sends a label), and the other simply branches.

In order to synthesize the type of the session offered by a classical process that matches this pattern we first we distinguish whether the branching is performed on the channel currently being synthesized ($c' = c$) or on another channel present in the linear context. In the former case, the process itself offers the choice. The function *synth_external_choice_c_proc_list* is invoked to typecheck each branch in l, producing a set of typed branch pairs and corresponding continuation processes. These are then assembled into a session type of the form *STExtChoice*, which represents the $A\&B$ session type, where the process offers multiple labelled continuations. The result is a tuple like in the previous examples, in this case the expanded process expression also carries the typing information of each branch (*proc_pairs*).

If instead $c' \neq c$, the choice occurs on a channel obtained from the linear context, meaning the current process is interacting with a session offered by another process. In this case the algorithm retrieves the session type associated with c' from the context and unfolds it in order to inspect its inner session type. The typing then proceeds only if the unfolded type is *STExtChoice*, ensuring that the channel's session type prescribes the correct choice behaviour. The helper function *synth_c_choice_from_other_proc_list* then checks that the labels and continuation processes in l are consistent with the branches

specified by the session type, updating the linear context and producing the resulting continuation type. In this situation the process consumes the choice offered on channel c', and the continuation types produced by the branches may themselves contain choices. These are then merged to construct the resulting session type of the case construct. If the channel does not exist in the context or does not correspond to a choice session type, an appropriate typing error is raised.

```
synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
  match proc with
  | CChoice (c', l) ->
    if
      c' = c then
        let lin_ctxt_opt, out_unr_ctxt, typed_pairs, proc_pairs, vars =
          synth_external_choice_c_proc_list lin_ctxt unr_ctxt env c l
            used_vars
        in
        begin match lin_ctxt_opt with
        | None -> error EmptyChoice
        | Some ctxt ->
            (ctxt, out_unr_ctxt, CChoice (c', proc_pairs),
              STExtChoice typed_pairs, vars)
        end
      else
        begin match List.assoc_opt c' lin_ctxt with
        | Some folded ->
            begin match unfold folded with
            | STExtChoice choice_pairs ->
                let lin_ctxt_opt, out_unr_ctxt, st_opt, new_procs, vars =
                  synth_c_choice_from_other_proc_list
                    lin_ctxt unr_ctxt env c c' l choice_pairs
                    used_vars
                in
                begin match lin_ctxt_opt with
                | None -> error EmptyChoice
                | Some ctxt ->
                    begin match st_opt with
                    | None -> error EmptyChoice
                    | Some st ->
                      (ctxt, out_unr_ctxt, CChoice (c', new_procs), st, vars)
                    end
                end
```

```
35              | wrong -> error (ChannelHasWrongSType (c', wrong))
36              end
37          | None -> error (NoSuchChannelInContext c')
38          end
39
40    | CLabel (c', l, p, _) ->
41        if c' = c then
42          let out_lin_ctxt, out_unr_ctxt, p', st, vars =
43            synth_class_proc unr_ctxt lin_ctxt env p c used_vars in
44          (out_lin_ctxt, out_unr_ctxt,
45            CLabel (c', l, p', Some st), STIntChoice [ (l, st) ], vars)
46        else
47          begin match List.assoc_opt c' lin_ctxt with
48          | Some folded ->
49              begin match unfold folded with
50              | STIntChoice choice_pairs ->
51                  begin match List.assoc_opt l choice_pairs with
52                  | None ->
53                    error (NoSuchLabelInType (l, STIntChoice choice_pairs))
54                  | Some st ->
55                      let new_lin_ctxt =
56                        (c', st) :: List.remove_assoc c' lin_ctxt in
57                      let out_lin_ctxt, out_unr_ctxt, p', st', vars =
58                        synth_class_proc unr_ctxt new_lin_ctxt env p c
59                          used_vars
60                      in
61                      (out_lin_ctxt, out_unr_ctxt,
62                        CLabel (c', l, p', Some st), st', vars)
63                  end
64              | wrong -> error (ChannelHasWrongSType (c', wrong))
65              end
66          | None -> error (NoSuchChannelInContext c')
67          end
```

If a process is matched to *CLabel* then we are selecting (choosing) which labelled branch should be executed. To type such a process we first need to understand if we are choosing on the channel offered by the process we are synthesizing or on a session offered by another process. If we are before the first scenario then the process is simply synthesized and we return the tuple containing the unrestricted context, the output linear context, the expanded process expression, the session type that encodes the choice that was made *STIntChoice[(l,st)]* (which represents the the $A_c \oplus B_c$ type in our syntax), and

the used variables list.

If the channel over which the process chooses is offered by another process we need to fetch the channelfrom the linear context, then unfold it to access the inner session type, which must be a choice offering a branch that matches our label. If the process offering the choice supports our labelled option we can update the linear context to reflect that the process will follow the behaviour associated with the chosen labelled option. Finally we synthesize the continuation process and return the correct tuple.

Below is the intuitionistic version of the same synthesis process to compare with the classical version described above.

```
synth_proc lin_ctxt env proc c used_vars =
  match proc with
  | Choice (c', l) -> (
      if
         c' = c then
        let ctxt_opt, typed_pairs, proc_pairs, vars =
          synth_external_choice_proc_list lin_ctxt env c l used_vars
        in
        match ctxt_opt with
        | None -> error EmptyChoice
        | Some ctxt ->
            (ctxt, Choice (c', proc_pairs), STExtChoice typed_pairs, vars)
      else
        match List.assoc_opt c' lin_ctxt with
        | Some folded -> (
            match unfold folded with
            | STIntChoice choice_pairs -> (
                let ctxt_opt, st_opt, new_procs, vars =
                  synth_internal_choice_proc_list
                    lin_ctxt env c c' l choice_pairs
                    used_vars
                in
                match ctxt_opt with
                | None -> error EmptyChoice
                | Some ctxt -> (
                    match st_opt with
                    | None -> error EmptyChoice
                    | Some st ->
                      (ctxt, Choice (c', new_procs), st, vars)))
            | wrong -> error (ChannelHasWrongSType (c', wrong)))
        | None -> error (NoSuchChannelInContext c'))
```

```
32
33    | Label (c', l, p, _) -> (
34        if c' = c then
35          let out_ctxt, p', st, vars =
36            synth_proc lin_ctxt env p c used_vars in
37          (out_ctxt, Label (c', l, p', Some st),
38            STIntChoice [ (l, st) ], vars)
39        else
40          match List.assoc_opt c' lin_ctxt with
41          | Some folded -> (
42              match unfold folded with
43              | STExtChoice choice_pairs -> (
44                  match List.assoc_opt l choice_pairs with
45                  | None ->
46                    error (NoSuchLabelInType (l, STExtChoice choice_pairs))
47                  | Some st ->
48                      let new_lin_ctxt =
49                        (c', st) :: List.remove_assoc c' lin_ctxt in
50                      let out_ctxt, p', st', vars =
51                        synth_proc new_lin_ctxt env p c used_vars
52                      in
53                      (out_ctxt, Label (c', l, p', Some st), st', vars))
54              | wrong -> error (ChannelHasWrongSType (c', wrong)))
55          | None -> error (NoSuchChannelInContext c'))
```

We will now introduce the process expression introduced with our work, and explain how these processes' type judgments are implemented in the typechecker. Starting with the simplest, the process denoting inaction *Terminate* (typed by *0* in the syntax):

```
1  synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
2    match proc with
3    | Terminate ->
4      if (List.length lin_ctxt <> 0) then
5        error (NonEmptyLinearContext)
6      else
7        (lin_ctxt, unr_ctxt, Terminate, STEnd, used_vars)
```

When a process layer expression is matched to this branch means we have finished executing every process and there are no channels left to close, and all the need to process needs to check is that linearity is indeed not broken by checking that the linear context is empty.

The processes that map to *Accept* and *Call* are processes that reintroduce weakening, allowing for session to be replicated, and thus used multiple times. In our system they represent servers and are synthesized as such:

```
synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
  match proc with
  | Accept (id, c', _, p) ->
      if c <> c' then
        error (MustAcceptOnOwnChannel c')
      else if id = c' then
        error (ChannelAlreadyExists id)
      else
        let (out_lin_ctxt, out_unr_ctxt, p', st, vars) =
          synth_class_proc unr_ctxt lin_ctxt env p id used_vars in
            if
              out_lin_ctxt = lin_ctxt
            then
              (out_lin_ctxt, out_unr_ctxt,
                Accept(id, c', Some st, p'), STSvrDefVar(st), vars)
            else
              error (LinearContextsMismatched (lin_ctxt, out_lin_ctxt))
```

The rule above types a process that promotes a linear session into an unrestricted channel, meaning it can then be replicated multiple times. First we must ensure that the channel that will be promoted is the the same channel being typed, otherwise we throw an error. Then we must ensure that the channel through which the client processes will communicate is not the same channel we are promoting. We can then synthesize the type of the process that will be run by the server, with which the clients interact. After that we need to ensure the input and output linear contexts are the same to ensure no channels are consumed incorrectly or leaked because after replication the linear context must remain the same, so that every communication protocol between processes remains consistent. If everything is correct we return a tuple with the output linear and unrestricted contexts along with the expanded process, the session type ($!A$) of the channel offered by the process, and the list of used variables. If the contexts do not match, we throw an error.

```
synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
  match proc with
  | Call (c', nc, _, p) ->
      if c' = nc then
        error (ChannelAlreadyExists nc)
      else if c' = c then
        error (CannotCallOwnChannel c')
```

```
8       else
9         begin match List.assoc_opt c' lin_ctxt with
10        | Some folded ->
11           begin match unfold folded with
12            | STClntInvVar st ->
13               let new_lin_ctxt = List.remove_assoc c' lin_ctxt in
14                 let new_unr_ctxt = (c', st) :: unr_ctxt in
15               begin match List.assoc_opt nc new_lin_ctxt with
16               | Some _ -> error (ChannelAlreadyExists nc)
17               | None ->
18                   begin match List.assoc_opt nc new_unr_ctxt with
19                   | Some _ -> error (ChannelAlreadyExists nc)
20                   | None ->
21                       let out_lin_ctxt, out_unr_ctxt, p', st', vars =
22                         synth_class_proc new_unr_ctxt
23                           ((nc, st) :: new_lin_ctxt) env p c used_vars
24                       in
25                         (out_lin_ctxt, out_unr_ctxt,
26                           Call (c' , nc, Some st, p'), st', vars)
27                   end
28               end
29           | wrong -> error (ChannelHasWrongSType (c', wrong))
30           end
31        | None ->
32           begin match List.assoc_opt c' unr_ctxt with
33           | Some folded ->
34             begin match unfold folded with
35             | stype ->
36               begin match List.assoc_opt nc lin_ctxt with
37               | Some _ -> error (ChannelAlreadyExists nc)
38               | None ->
39                   begin match List.assoc_opt nc unr_ctxt with
40                   | Some _ -> error (ChannelAlreadyExists nc)
41                   | None ->
42                       let out_lin_ctxt, out_unr_ctxt, p', st', vars =
43                         synth_class_proc unr_ctxt
44                           ((nc, stype) :: lin_ctxt) env p c used_vars
45                       in
46                         (out_lin_ctxt, out_unr_ctxt,
47                           Call (c', nc, Some stype, p'), st', vars)
48                   end
```

```
49                end
50              end
51            | None -> error (NoSuchChannelInContext c')
52            end
53        end
```

The dual operation *Call* is synthesized by the algorithm presented above. The channel *nc* must be a fresh session, as such we start the typing process by checking that the session is different from the process' own session, if it is the same we throw an error to prevent aliasing. Then we check that we are not trying to invoke a client session on the process' offered channel, this would mean that we are invoking a service offered by the process itself, which is not permitted. After that we must ascertain in which context the session resides. We start by searching the linear context, where if found, the session must be typed as *STClntInvVar*, corresponding to *?A* (after unfolding, in case it is recursively typed), meaning that it can offer multiple client invocations. Next we move the session from the linear context to the unrestricted context where it will now be typed simply as A, meaning the session is consumed in the linear context (removing it from said context) to now be used multiple times. This is needed because a server can be invoked multiple times by different clients, but every linear resource must be consumed for a program to be well typed.

We then check that the channel, *nc*, that will be used to interact with the server does not already exist in any context (it must be a fresh session as previously mentioned). If it is not present in any context then we proceed to add it to the linear context, typed with the same session type as the server *st* in order to match the behaviour of said server, and then proceed to synthesize the continuation p, returning a tuple containing the output contexts, the annotated process expression, the session type of the process' offered channel, and a list of used variable names.

In the case that the session channel c' was already present in the unrestricted context, then the session was already promoted previously, meaning all that is now needed to correctly type this session is to ensure that the new channel *nc* is not present in any context, and then add it to the linear context with the same session type presented by c', proceeding then with the synthesis of the continuation, and returning the exact same tuple.

Process expression of the form *Affine* promote a resource to an affine session, where it can now be used or discarded arbitrarily (but only once). In order to type this promotion correctly we first need to ascertain if we're promoting the offered session or a session being offered by another process.

```
1  synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
2    match proc with
3    | Affine (c', deps, _, p) ->
4        if c' = c then
```

```
5           let out_lin_ctxt, out_unr_ctxt, p', st', vars =
6                 synth_class_proc unr_ctxt lin_ctxt env p c used_vars
7           in
8             begin match
9               (try_collect_discardable_deps deps lin_ctxt unr_ctxt)
10            with
11            | Some new_deps ->
12              (out_lin_ctxt, out_unr_ctxt, Affine (c', new_deps, Some st', p'),
13                STAffine (st', new_deps), vars)
14            | None -> error(HasNonDisposableDependecies (c', deps))
15            end
16        else
17          begin match List.assoc_opt c' lin_ctxt with
18          | Some folded ->
19            begin match unfold folded with
20            | STAffine _ ->
21                begin match
22                  (try_collect_discardable_deps deps lin_ctxt unr_ctxt)
23                with
24                | Some new_deps ->
25                    let out_lin_ctxt, out_unr_ctxt, p', st', vars =
26                      synth_class_proc unr_ctxt lin_ctxt env p c used_vars
27                    in
28                      (out_lin_ctxt, out_unr_ctxt,
29                        Affine (c', new_deps, Some st', p'), st', vars)
30                | None -> error (HasNonDisposableDependecies (c', deps))
31                end
32            | wrong_st -> error (ChannelHasWrongSType (c', wrong_st))
33              end
34          | None ->
35            error (NoSuchChannelInContext c')
36          end
```

If it is the first case then we immediately synthesize the continuation process so that we can then understand which channels, present in the continuation, will depend on the session being promoted to affine, and if those dependencies are *discardable*, meaning they can remain unused, for example any resource in the unrestricted context would be considered discardable. This is done with the help of the function *try_collect_discardable_deps*, which is presented below. This function also normalizes the dependency list into a list that contains the name of the dependency as well as the evaluated session type option. Finally we return a tuple with all the normal information (the session type return is updated to

also reference the normalized dependency list).

If the session is being offered by another process, it must then be present in the linear context with the *STAffine* (in the syntax $\wedge A$) session type, meaning the process offering the session is trying to either use or discard the session, meaning we must promote it first. We then proceed by verifying that every dependency is indeed discardable, and then we synthesize the continuing process p, returning the correct tuple.

```
try_collect_discardable_deps deps lin_ctxt unr_ctxt =
  let rec aux_fun acc l =
    begin match l with
    | [] -> Some (List.rev acc)
    | (dep, _) :: rest ->
      begin match List.assoc_opt dep lin_ctxt with
      | Some ((
        STAffine _
        | STCoAffine _
        | STFullUse _) as stype) ->
          aux_fun ((dep, Some stype) :: acc) rest
      | _ ->
        begin match List.assoc_opt dep unr_ctxt with
        | Some stype ->
          aux_fun ((dep, Some stype) :: acc) rest
        | None -> None
        end
      end
    end
  in aux_fun [] deps
```

To interact with an *affine* session a process must either *use* or *discard* the session. The first case is mapped next:

```
synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
  match proc with
  | Use (c', _, p) ->
      if c' = c then
        let out_lin_ctxt, out_unr_ctxt, p', st', vars =
          synth_class_proc unr_ctxt lin_ctxt env p c used_vars
        in
          (((c', STCoAffine (st', [])) :: List.remove_assoc c' out_lin_ctxt),
            out_unr_ctxt, Use (c', Some st', p'), STCoAffine (st', []), vars)
      else
      begin match List.assoc_opt c' lin_ctxt with
```

```
12        | Some folded ->
13          begin match unfold folded with
14          | STCoAffine (st, _) ->
15              let out_lin_ctxt, out_unr_ctxt, p', st', vars =
16                synth_class_proc unr_ctxt
17                  ((c', st) :: (List.remove_assoc c' lin_ctxt)) env p c
18                  used_vars
19              in
20                (out_lin_ctxt, out_unr_ctxt, Use(c', Some st', p'), st', vars)
21          | wrong_st -> error (ChannelHasWrongSType (c', wrong_st))
22          end
23        | None -> error (NoSuchChannelInContext c')
24        end
```

To correctly synthesize the type of a *Use* session we start by checking if the session is offered by the process or provided by another process entirely. If the process itself is issuing the usage of an affine session, meaning the process wishes to utilize the session for some purpose then we must synthesize the type of the continuation (where the channel must be used), after that we return the output linear context, updating it by wrapping the offered session c' in the *STCoAffine* type (equivalent to $\vee A$).

If on the other hand the process is using an *affine* session offered by some other process, then we must get the channel from the linear context, where it must be typed as *STCoAffine*. We then update the channels type in the linear context and proceed to synthesize the continuation process, returning the tuple with the output contexts, the annotated process expression, the type of the offered session and the list of used variables.

We can alternatively *discard* the affine session. To type such a session we first need to ensure that the session being discarded is not the one offered by the process itself, since that would result in early, wrongful termination. After that we acquire the channel from the linear context, where we expect it to have *STCoAffine* type, throwing an error if that is not the case. If the channel is in the context with the correct type associated, then we proceed to remove the session c' from the input linear context of the continuing process, before synthesizing its type.

```
1  synth_class_proc unr_ctxt lin_ctxt env proc c used_vars =
2    match proc with
3    | Discard (c', _, p) ->
4      if c' = c then error (CannotDisposeOwnChannel c');
5      begin match List.assoc_opt c' lin_ctxt with
6      | Some folded ->
7        begin match unfold folded with
8        | STCoAffine _ ->
```

```
        let out_lin_ctxt, out_unr_ctxt, p', st', vars =
          synth_class_proc unr_ctxt
            (List.remove_assoc c' lin_ctxt) env p c
            used_vars
        in
          (out_lin_ctxt, out_unr_ctxt,
            Discard(c', Some st', p'), st', vars)
      | wrong_st -> error (ChannelHasWrongSType (c', wrong_st))
      end
    | None -> error (NoSuchChannelInContext c')
    end
```

Our language also supports processes that model mutually exclusive, shared memory cells. To create such a cell a process must express itself as a *Cell* expression and we type it as such:

```
| Cell (c', _, a, q) ->
    let out_lin_ctxt, out_unr_ctxt, q', st', vars =
      synth_class_proc unr_ctxt lin_ctxt env q a used_vars
    in
      begin match st' with
      | STAffine (stype, _) ->
          if c' = c then
            (out_lin_ctxt, out_unr_ctxt,
              Cell(c', Some stype, a, q'), STFullState(stype), vars)
          else
            begin match List.assoc_opt c' lin_ctxt with
            | Some folded ->
              begin match unfold folded with
              | STFullState (st) ->
                let dual_st = construct_dual_stype st env in
                if
                  (subtyping [] dual_st stype
                    || subtyping [] stype dual_st)
                then
                  (out_lin_ctxt, out_unr_ctxt,
                    Cell(c', Some stype, a, q'), STFullState(stype), vars)
                else error (ChannelHasWrongSType (c', dual_st))
              | wrong -> error (ChannelHasWrongSType (c', wrong))
              end
            | None -> error (NoSuchChannelInContext c')
```

```
            end
        | wrong -> error (ChannelHasWrongSType (a, wrong))
        end
```

We begin by synthesizing the process expression that is being "stored" inside the cell q. We then check that the resulting session type st' is *STAffine*, meaning it can (at any point) be discarded. If it is then we check if the session c' that will be used for communicating with the cell itself is offered by the process, in which case we return a tuple, or not. The returned tuple contains the output contexts obtained from the synthesis of the stored process, the annotated process expression (which records what type of information the cell stores *stype*), the type of the session, which is obtained by wrapping the cell storage type *stype* with the *STFullState* type, which represents a cell in the "full state" (in the syntax $S_{\bullet}A$) and finally the list of used variables.

If on the other hand the channel c' is being offered by some other process, then we must first fetch it from the linear context, where it must be present with the correct type (*STFullState*). Next we can construct the dual of the type of the session offered by the process interacting with the cell, in order to ensure we follow the correct protocol. If these sessions' types are not dual then it means one process is trying to interact with the stored process with a mismatched or wrong protocol. Finally we return the same tuple as before.

A process can also model an empty cell, meaning it models a cell that, currently holds no information. To do so the expression must match the *Empty* branch for which we next describe the typing procedure:

```
| Empty (c', st) ->
  if c' = c then
    (lin_ctxt, unr_ctxt, Empty(c', st), STEmptyState (st), used_vars)
  else
    begin match List.assoc_opt c' lin_ctxt with
    | Some folded ->
      begin match unfold folded with
      | STEmptyState (stype) ->
        let dual_st = construct_dual_stype stype env in
        if (subtyping [] dual_st st || subtyping [] st dual_st) then
        ((c', stype) :: (List.remove_assoc c' lin_ctxt), unr_ctxt,
          Empty(c', st), STEmptyState (st), used_vars)
        else error (ChannelHasWrongSType (c', dual_st))
      | wrong -> error (ChannelHasWrongSType (c', wrong))
      end
    | None -> error (NoSuchChannelInContext c')
    end
```

This is an example of a process which must have a type annotation, in this case specifying what type of values the cell can store, before we can synthesize the type of the session being offered. As such, if the session is being offered on the process itself we simply return the tuple with the contexts (untouched), the already annotated process expression, the tpye of the session, which is just the type of the values the cell can store *st* wrapped with the session type that represents an empty cell *STEmptyState* (corresponding to $S \circ A$), and the list of used variables (also unmodified).

If the session is being offered by some other process, then we need to grab it from the linear context, where it must be typed (after unfolding to deal with the possibility of recursion) *STEmptyState* and the session type *stype* must be the dual of the type of the session of the process the cell will store. After that we can update the session type of the channel c' present in the linear context and return the same tuple as before.

We can interact with a cell in three ways, either by grabbing the session being stored (done through *Take*), by repopulating and empty cell with another process expression (using *Put*), or releasing a process' reference to the cell (with *Release*). Next we explain how these three processes are typed:

```
1   | Take (c', opt, a, p) ->
2       if c' = c then
3         begin match opt with
4         | Some st ->
5           let out_lin_ctxt, out_unr_ctxt, p', st', vars =
6             synth_class_proc unr_ctxt
7               ((a, STCoAffine(st, [])) :: lin_ctxt)
8               env p c used_vars
9           in
10              begin match List.assoc_opt c' out_lin_ctxt with
11              | Some folded_st ->
12                begin match unfold folded_st with
13                | STEmptyUse (stype) ->
14                    if
15                      (subtyping [] st stype || subtyping [] stype st)
16                    then
17                      ((List.remove_assoc c' out_lin_ctxt), out_unr_ctxt,
18                        Take(c', Some st', a, p'), STFullUse(st'), vars)
19                    else error (ChannelHasWrongSType (a, st'))
20                | wrong -> error (ChannelHasWrongSType (c', wrong))
21                end
22              | None -> error (NoSuchChannelInContext c')
23              end
24          | None -> error (CannotInferType)
```

```
25          end
26        else
27          begin match List.assoc_opt c' lin_ctxt with
28          | Some folded ->
29            let unfolded = unfold folded in
30              begin match unfolded with
31              | STFullUse (stype) ->
32                let new_lin_ctxt =
33                  (c', STEmptyUse (stype))
34                  ::
35                  (List.remove_assoc c' lin_ctxt) in
36                let out_lin_ctxt, out_unr_ctxt, p', st', vars =
37                  synth_class_proc unr_ctxt
38                    ((a, STCoAffine (stype, [])) :: new_lin_ctxt)
39                    env p c used_vars
40                in
41                  (out_lin_ctxt, out_unr_ctxt,
42                    Take(c', Some stype, a, p'), st', vars)
43              | wrong -> error (ChannelHasWrongSType (c', wrong))
44              end
45          | None -> error (NoSuchChannelInContext c')
46          end
```

To type a *Take* process we first check if the session we're synthesizing is the one offered by the process. If it is then an annotation must be included in the process expression identifying the type of the acquired session (a), which will eventually be used to interact with the process that is being stored. After that we add it to the linear context and proceed to synthesize the continuation p.

As explained in Section 3.2.2 we must ensure that after every *take*, a *put* expression exists. This is done next by checking the linear context outputted by the synthesis of the continuation p, and ensuring there is a reference to offered channel c' with the correct session type of *STEmptyUse* (equivalent to $U_{\circ}A$ in the syntax). If this is the case, we then check that we repopulated correctly (with the appropriate session type) and proceed to return a tuple with the output linear context (without the reference to the session channel), the output unrestricted context, the annotated process expression, the session type that describes the interaction with the cell st' wrapped by the *STFullUse* type to properly type the session being offered here, and finally the list of used variable names.

If the channel c' is instead offered by another process, then we simply fetch the channel from the linear context, unfold any possible recursive session type, and verify that the innermost session type conforms to *STFullUse* (which models $U_{\bullet}A$ in the syntax). If this is the case we update the session's type in the linear context to be of type *STEmptyUse*

so that when we synthesize the continuation p we force a *Put* expression to be present to consume the channel. After the synthesis procedure we return the tuple containing both output contexts, the annotated process expression, the session type of the offered endpoint, and the list of used variables.

```
| Put (c', _, a, q, p) ->
  if c' = c then
    let _, _, q', q_st', _ =
    synth_class_proc unr_ctxt lin_ctxt env q a used_vars
    in
      let p_out_lin, p_out_unr, p', p_st', p_vars =
        synth_class_proc unr_ctxt lin_ctxt env p c used_vars
      in
        begin match q_st' with
        | STAffine (st1, _) ->
          begin match p_st' with
          | STFullUse (st2) ->
            let dual_st = construct_dual_stype st2 env in
            if
              (subtyping [] st1 dual_st || subtyping [] dual_st st1)
            then
              ((c', STEmptyUse (st1))
                ::
                (List.remove_assoc c' p_out_lin), p_out_unr,
                Put(c', Some st1, a, q', p'), STEmptyUse (st1), p_vars)
            else error (ChannelHasWrongSType (a, q_st'))
            | c'_wrong -> error (ChannelHasWrongSType (c', c'_wrong))
          end
        | a_wrong -> error (ChannelHasWrongSType (a, a_wrong))
        end
  else
    begin match List.assoc_opt c' lin_ctxt with
    | Some folded ->
      begin match unfold folded with
      | STEmptyUse (st1) ->
          let p_out_lin, p_out_unr, p', p_st', p_vars =
            synth_class_proc unr_ctxt
              ((c', STFullUse (st1)) :: (List.remove_assoc c' lin_ctxt))
              env p c used_vars
          in
            let _, _, q', q_st', _ =
```

```
37                synth_class_proc unr_ctxt lin_ctxt env q a used_vars
38              in
39              begin match q_st' with
40              | STAffine (st2, _) ->
41                let dual_st = construct_dual_stype st2 env in
42                if
43                  (subtyping [] dual_st st1 || subtyping [] st1 dual_st)
44                then
45                  (p_out_lin, p_out_unr,
46                    Put(c', Some st1, a, q', p'), p_st', p_vars)
47                else error (ChannelHasWrongSType (a, dual_st))
48              | wrong_st -> error (ChannelHasWrongSType (a, wrong_st))
49              end
50          | wrong -> error (ChannelHasWrongSType (c', wrong))
51          end
52        | None -> error (NoSuchChannelInContext c')
53        end
```

The expression typed above is used for repopulating the cell with a process expression. First we check if the offered session is the same as the one being synthesized. If it is then we proceed to synthesize both the process expression that will repopulate the cell q and the continuation of this process p. Next we ensure that the session offered by the cell's new process is *affine*, that the continuation p correctly types to *STFullUse* (meaning we are still interacting with the cell), and that both processes' sessions *st1* and *st2* have dual session types (lines 14-15). This last verification is important because the continuation might eventually interact with the stored process, and thus needs to have the correct session protocol. Finally a tuple is returned with the output linear context, completed with the updated reference to the offered session c' to complete the expectations mentioned before in *Take*, the output unrestricted context, the annotated process expression, the composite session type *STEmptyUse(st1)*, which represents a *put* expression that placed a process offering a session typed *st1* in a cell, and finally the list of unused variables.

If on the other hand the offered session is offered by some other process we search for it in the linear context, checking that it has the correct session type (*STEmptyUse*). We then proceed to synthesize the continuation, updating the reference to c' in the linear context to have the correct type of *STFullUse*. Next we proceed as before, synthesizing the type of the session offered by the process being stored in the cell, checking that it is an *affine* process, and that *st1* and *st2*, the session offered by the continuations are duals of each other (lines 34-35). We also return the same tuple as before, with the exception of the output linear context of the continuation process p, which is directly returned without modifications.

To *release* a reference to a cell, the process must match the expression *Release*. To synthesize this expression we first check to see if the offered session is the one being

synthesized, if that's the case we return an error. Otherwise we simply fetch the session from the linear context, check that it has the appropriate type (*STFullUse*, since we can only release cells in the unlocked state $U_\bullet A$). We then proceed to remove the session c' from the linear context, thus releasing it, and finally we synthesize the continuation p, returning the same tuple as all other operations.

```
| Release (c', p) ->
  if c' = c then
    error (CannotDisposeOwnChannel c')
    else
    begin match List.assoc_opt c' lin_ctxt with
    | Some folded ->
      begin match unfold folded with
      | STFullUse _ ->
        let new_lin_ctxt = List.remove_assoc c' lin_ctxt in
          let out_lin_ctxt, out_unr_ctxt, p', st', vars =
          synth_class_proc unr_ctxt new_lin_ctxt env p c used_vars in
            (out_lin_ctxt, out_unr_ctxt, Release(c', p'), st', vars)
      | wrong -> error (ChannelHasWrongSType (c', wrong))
      end
    | None -> error (NoSuchChannelInContext c')
    end
```

To allow a cell to be shared between multiple process we utilize the process expression *Share*, which is typed like so:

```
| Share (c', _, q, r) ->
  if c' = c then
    let q_out_lin, q_out_unr, q', q_st', q_vars =
      synth_class_proc unr_ctxt lin_ctxt env q c used_vars
    in
      let r_out_lin, r_out_unr, r', r_st', r_vars =
        synth_class_proc unr_ctxt lin_ctxt env r c used_vars
      in
        begin match (q_st', r_st') with
          | (STFullUse(st1), STFullUse(st2)) ->
            if (subtyping [] st1 st2 || subtyping [] st2 st1) then
                (q_out_lin @ r_out_lin, q_out_unr @ r_out_unr,
                  Share(c', Some st1, q', r'), r_st', q_vars @ r_vars)
            else error (AllCasesMustProduceIdenticalType (st1, st2))
          | (STFullUse(st1), STEmptyUse(st2))
          | (STEmptyUse(st1), STFullUse(st2)) ->
```

```
17              if (subtyping [] st1 st2 || subtyping [] st2 st1) then
18                (q_out_lin @ r_out_lin, q_out_unr @ r_out_unr,
19                  Share(c', Some st1, q', r'), r_st', q_vars @ r_vars)
20              else error (AllCasesMustProduceIdenticalType (st1, st2))
21            | (wrong1, wrong2) -> error (ShareNotAllowed (wrong1, wrong2))
22            end
23      else
24        let q_out_lin, q_out_unr, q', q_st', q_vars =
25          synth_class_proc unr_ctxt lin_ctxt env q c used_vars
26        in
27          let r_out_lin, r_out_unr, r', r_st', r_vars =
28            synth_class_proc unr_ctxt lin_ctxt env r c used_vars
29          in
30          begin match List.assoc_opt c' lin_ctxt with
31          | Some folded ->
32            begin match unfold folded with
33            | STFullUse st ->
34              if (subtyping [] q_st' r_st' || subtyping [] r_st' q_st') then
35                (q_out_lin @ r_out_lin, q_out_unr @ r_out_unr,
36                  Share(c', Some (STFullUse(st)), q', r'),
37                  r_st', q_vars @ r_vars)
38              else error (AllCasesMustProduceIdenticalType(q_st', r_st'))
39            | wrong -> error (ChannelHasWrongSType (c', wrong))
40            end
41          | None -> error (NoSuchChannelInContext c')
42          end
```

First we check if the session being shared is the one being synthesized, if it is we synthesize both *branches* of the *share* process, which correspond to the processes that will share access to the cell. After synthesis, we match the types of the sessions offered to verify they are valid, according to the typing judgments for *Share*, listed in Section 3.2.2, and shown above. If everything is correct, meaning the synthesized session types *q_st'* and *r_st'* match a valid variation of *STFullUse* and *STEmptyUse* we proceed to check if they then behave according to the correct protocol, which implies the session offered by their continuations has the same type (lines 12 and 16). Finally we return a tuple with the concatenated output linear context, the concatenated unrestricted context, the annotated process expression, the synthesized session type, and the list of used variables.

If the session is being offered by another process, then we obtain the session's type by getting the session from the linear context and unfolding any possible recursion it might have. If the session is not present in the linear context an error is returned. Then if the session is of type *STFullUse* we proceed with the synthesis of the two branching processes,

and if they both produce the same session type (to ensure consistency) then we return the same tuple as before.

4.2 Compilation

Although compiling to *Go* stems from using J. Geraldo's work as the basis for our implementation, the advantages offered by this target language are significant enough that we would likely choose *Go* regardless. In particular, *Go* provides efficient channel-based concurrency primitives and lightweight threads (*goroutines*), which align well with the communication-oriented nature of our language. Additionally, *Go*'s mature compiler infrastructure and compile-time optimizations allow us to leverage existing engineering efforts rather than implementing these mechanisms from scratch.

Compiling functions and processes is generally straightforward. Specifically, a process construct of the form $c, e_1 ... e_n \leftarrow \{P\} \leftarrow d_1 ... d_n$ (or it's intuitionistic variant) in our language translates into a *Go* function with parameters corresponding to the channels listed in $d_1 ... d_n$, $e_1 ... e_n$ and c. Functional definitions are compiled through a direct correspondence to standard Go functions. By contrast the compilation of channel forwarding requires a more involved translation, which is discussed in detail in [14]

Using *Go* as the target language means our compilation strategy must address a mismatch between the typing discipline of channels in *Go* and that of our session-typed language. In *Go*, channels have a fixed payload type determined at creation time, whereas session-typed channels evolve as the communication protocol progresses. For instance, a session of type $int \wedge bool \wedge 1$ requires the exchange of an integer, followed by a boolean value, and finally a termination signal, meaning that the type of messages communicated over the channel changes over time. Such behaviour cannot be represented directly using a single *Go* channel. To bridge this gap, we exploit the type information produced during the typechecking phase and stored in the abstract syntax trees to encode a single session channel as a collection of *Go* channels. This approach was already used in Geraldo's work [14] where it is discussed in much more detail.

Building on this encoding strategy, it is worth noting that there are two distinct approaches for translating a session type AST into a collection of Go channels. The single-channel approach generates a single Go channel of type `interface{}`, which is then cast to the appropriate session type representation at each usage. In contrast, the multi-channel approach creates multiple Go channels, one for each compiled type (state). The operational logic for processes is essentially the same in both approaches; the main differences concern channel typing, generation, and concurrency. For clarity, we focus on examples using the single-channel approach.

Another constraint imposed by *Go* is that every variable must be utilized. In our language however, some values can be safely ignored, for example, a received value may be discarded without further action. Following the approach introduced by J. Geraldo

and adopted in our work, the typechecking process tracks all used values, and any value found to be unused is replaced by the *blank identifier* ("_").

Because session types naturally define state-like communication protocols, where each interaction advances the session to a new state, we model session channels in the generated code as a sequence of protocol states. Each session state is represented by a custom *Go* type (*structs*) which is accompanied by some additional methods that implement the corresponding communication operations. *States* will have one of the following types:

- Communication (*Comm*) - encodes data exchange session types;
- Choice (*Choice*) - encodes choice session types;
- Terminal (*End*) - encodes terminal session types;
- Server (*Server*) - encodes the !*A* session type;
- Affine (*Aff*) - encodes the behaviour of an affine state session type;
- Cell (*Cell*) - encodes the behaviour of shared mutable cell;

For communication states, the generated type contains the underlying channel, the type of the communicated value, and a reference to the next protocol state. Choice session types instead maintain a mapping from labels to their respective continuation states, allowing the protocol to branch depending on the exchanged label. Terminal session types represent the end of the protocol and therefore have no successor state. For server states the reference to the next state points to themselves, basically emulating a looped communication state. Affine states have a channel and a reference to the successor state, they also maintain a boolean flag which marks them as discarded or not, and a list of state names that correspond to the dependencies of the affine state. Because these dependencies must all be "discardable", all of them also need to implement an interface that implements a *Dispose()* method. Finally we have the cell state, which is by far the most complex. It is composed of a channel, a closure that models a slot for storing processes, a reference to the state that implements that process, a mutex lock and a conditional variable for goroutine coordination, along with a couple of boolean flags and counters to coordinate clean up. Although not all session states are explicitly written in the source program, they are reconstructed during type checking and incorporated into the abstract syntax tree, enabling the compiler to generate the corresponding protocol representations.

The compilation of a program proceeds in two main stages: *preamble generation* and *code generation*. The process' entry point is the function `compile_prog`, which manages the translation of the source program into Go code. During the preamble generation phase, the compiler maps the session types to the states previously mentioned, keeping these structures in a map. To build this map the compiler takes into account both explicit declarations of our program as well as the main expression to be executed. Each state is represented as a structured object consisting of a unique identifier and a body. This

structure allows recursive session types to be handled safely by creating dummy states and references initially, which are later updated once the recursive cycle is unfolded, thus ensuring a correct representation of cycles within the session protocol. Identifiers for states are generated sequentially, guaranteeing uniqueness and consistency across the compiled code.

The function responsible for constructing this type-state association map is called *make_state_trees_from_exp* and has the following signature:

```
let rec make_state_trees_from_exp map e : state TypeTable.t ->
  exp -> state TypeTable.t
```

The function takes a *map* that associations between session types and their respective state representation. Each entry has as a key the session type, which means that structurally equal session types, like recursive types, are treated as the same key for this map, and thus are linked to a single state. This map is initially passed as an empty map, and during the exploration of the Abstract Syntax Trees (ASTs) of every declaration and the main expression present in the program, we call *make_state* for every session type found, which creates the corresponding state.

```
let rec make_state map env stype : state TypeTable.t ->
  (var * state ref) list -> stype -> state TypeTable.t
```

Above is the signature of the state building function, and bellow we present a couple of relevant examples of the process of translating a session type into a state.

```
let rec make_state map env stype =
match get_state map stype with
| Some _ -> map
| None ->
  match stype with
    | STSend (t, st) -> (
        let state_id = TypeStore.get_id stype in
        match st with
        | STRec (v, _) -> (
            match List.assoc_opt v env with
            | Some dummy_ref ->
                let this_state =
                  State (state_id, Comm (t, dummy_ref)) in
                TypeTable.replace map stype this_state;
                map
            | None ->
                let new_map = make_state map env st in
```

```
                let this_state =
                  State (state_id,
                    Comm (t, ref (TypeTable.find new_map st)))
                in
                TypeTable.replace new_map stype this_state;
                new_map)
          | _ ->
              let new_map = make_state map env st in
              let this_state =
                State (state_id, Comm (t, ref (TypeTable.find new_map st)))
              in
              TypeTable.replace new_map stype this_state;
              new_map)

  | STClntInvVar st | STSvrDefVar st ->
        let state_id = TypeStore.get_id stype in
          begin match st with
          | STRec (v, _) ->
              begin match List.assoc_opt v env with
              | Some dummy_ref ->
                  let this_state = State (state_id,
                    Server(TClassProc(st, [], []), dummy_ref)) in
                  TypeTable.replace map stype this_state;
                  map
              | None ->
                let new_map = make_state map env st in
                  let
                    this_state = State (state_id,
                      Server(TClassProc(st, [], []),
                      ref (TypeTable.find new_map st)))
                  in
                    TypeTable.replace new_map stype this_state;
                    new_map
              end
          | _ ->
              let new_map = make_state map env st in
                    let
                      this_state = State (state_id,
                        Server(TClassProc(st, [], []),
                        ref (TypeTable.find new_map st)))
                    in
```

```
28                    TypeTable.replace new_map stype this_state;
29                    new_map
30            end
```

Above are the protocols for transforming a session type of *STSend*, *STSvrDefVar* and *STClntInvVar* into their respective *Communication*, and *Server* states.

```
1   | STAffine (st, deps) | STCoAffine (st, deps) ->
2           let state_id = TypeStore.get_id stype in
3             let next_map, deps_list = make_deps_list map env deps in
4             begin match stype with
5             | STRec (v, _) ->
6                 begin match List.assoc_opt v env with
7                 | Some dummy_ref ->
8                     let this_state = State(state_id,
9                       Aff(TClassProc(st, [], []), dummy_ref, deps_list)) in
10                    TypeTable.replace next_map stype this_state;
11                    next_map
12                | None ->
13                  let new_map = make_state next_map env st in
14                    let this_state =
15                      State (state_id, Aff(TClassProc(st, [], []),
16                        ref (TypeTable.find new_map st), deps_list))
17                    in
18                      TypeTable.replace new_map stype this_state;
19                      new_map
20                end
21            | _ ->
22              let new_map = make_state next_map env st in
23                let this_state =
24                  State (state_id, Aff(TClassProc(st, [], []),
25                    ref (TypeTable.find new_map st), deps_list))
26                in
27                  TypeTable.replace new_map stype this_state;
28                  new_map
29            end
```

For both *STAffine* and *STCoAffine* session types we produce the same state, as shown above. For the session types that shared memory cells and their operations we also utilize a single state, this is exemplified below:

```
1       | STFullState (stored_st) | STFullUse (stored_st)
2       | STEmptyState (stored_st) | STEmptyUse (stored_st) ->
```

```
3           needs_sync := true;
4           let state_id = TypeStore.get_id (STFullState(stored_st)) in
5             let affine_st = STAffine(stored_st, []) in
6               let new_map = make_state map env affine_st in
7                 let this_state = State(state_id,
8                   Cell(TClassProc(affine_st, [], []),
9                   ref (TypeTable.find new_map affine_st))) in
10                  TypeTable.replace new_map (STFullState(stored_st))
11                  this_state;
12                  new_map
```

Once the state map is established, the compiler generates Go code templates corresponding to each session state. Each state is encoded as a *Go* struct, whose fields capture the runtime components required by the protocol and whose methods implement the behaviour associated with that state. As the structure of these states was described previously, we focus here on the methods that accompany them. Communication states provide *Send* and *Recv* methods implementing the corresponding communication actions. Choice states similarly expose *Send* and *Recv* methods through which the branch label is exchanged. Terminal states implement minimal *Send* and *Recv* methods used to conclude the session.

Server states provide two methods: *Call*, which receives from the server channel a fresh channel from a client through which subsequent communication occurs, and *Accept*, which returns the received channel through the server channel to establish the interaction. Affine states provide several methods that enforce the affine usage discipline. The *Use* method, sends a *use* decision through the state's channel, and returns a reference to the next protocol state. The *AwaitDecision* method blocks until a message is received on the channel, ignoring the transmitted value but returning the boolean result of the receive operation together with a reference to the successor state. Affine states also implement a *Dispose* method which guarantees that disposal occurs at most once, and recursively disposes of all dependent states, finally closing the state's underlying channel.

Cell states provide a richer interface consisting of the following methods:

- *Release*, which locks the state, increments the counter of completed operations, and determines whether the cell cleanup procedure should be triggered.
- *Dispose*, which performs the cell's clean up, discarding it and triggering the disposal of the stored process and its corresponding state.
- *Take*, which implements the behaviour of the *take* process.
- *Put*, which implements the behaviour of the *put* process.

Following preamble generation, the code generation phase translates expressions, function definitions, and processes. Each process is compiled with careful tracking of state

identifiers to maintain the linear progression of the session protocol. Spawned processes are initialized according to their session type and executed in goroutines, with proper argument passing and continuation compilation.

The compilation of declarations and expressions is driven by a few core functions. Declarations can be either functions or type definitions, so their compilation result is either a Go function as a string or an empty string. Key functions in this phase include:

```
let rec compile_type type_map ty : state TypeTable.t -> ty -> string
```

This function takes the type-state map and a type to compile, returning a string representing the corresponding Go type.

```
let rec compile_return_exp type_map exp env : state TypeTable.t ->
  exp -> exp SeqMap.t -> string
```

This function compiles an expression intended as a function return value, prepending the resulting code with *return*. The function shown below differs (in behaviour) from this one only in that it compiles an expression that does not need to return, including the main expression of the program.

```
let rec compile_exp type_map exp env is_rec : state TypeTable.t ->
  exp -> exp SeqMap.t -> bool -> string
```

To compile processes in our language we use two functions: one for intuitionistic processes *compile_proc*, which compiles a process expression, taking the type-state mapping, the process to compile, a mapping of process identifiers to state identifiers, a functional environment and a boolean that allows us to distinguish between recursive and non-recursive processes; and another, *compile_c_proc*, for classical processes which behaves in much the same way, but for classical processes, which will result in different compilation results.

```
let rec compile_proc type_map p id_map env is_rec : state TypeTable.t ->
  proc -> int ref SeqMap.t -> exp SeqMap.t -> bool -> string
```

```
let rec compile_c_proc type_map p id_map env is_rec : state TypeTable.t ->
  c_proc -> int ref SeqMap.t -> exp SeqMap.t -> bool -> string
```

- `compile_proc(type_map, p, id_map, env, is_rec)`: C
- `compile_c_proc(type_map, p, id_map, env, is_rec)`: Compiles a process expression, taking the type-state mapping, the process to compile, a mapping of process identifiers to state identifiers, and a functional environment.

For example, a *CSend* process is compiled as:

```
compile_c_proc type_map p id_map env is_rec =
  match p with
  | CSend (d, exp, _, proc) ->
      let curr_id = curr_seq_id d id_map in
      let next_id, next_map = next_seq_id d id_map in
      Printf.bprintf !compiled "%s := %s.Send(" next_id curr_id;
      compile_exp type_map exp env is_rec;
      Printf.bprintf !compiled ")\n";
      compile_c_proc type_map proc next_map env is_rec
```

While a *CSpawn* process is compiled as:

```
  | CSpawn (d, e, opt, p, unr_args, lin_args) ->
      begin match opt with
      | Some st ->
          begin match get_state type_map st with
          | Some (State (id, _)) ->
            begin match
              (List.length lin_args = 0, List.length unr_args = 0)
            with
            | (true, true) ->
                init_state_template d id;
                if(!needs_sync) then
                  (Buffer.add_string !compiled "wg.Add(1)\n ";);
                Buffer.add_string !compiled "go func(){\n ";
                if(!needs_sync) then
                  (Buffer.add_string !compiled "defer wg.Done()\n ";);
                compile_exp type_map e env is_rec;
                Printf.bprintf !compiled "(%s)\n}()\n" d;
                compile_c_proc type_map p id_map env is_rec
            | (true, false) ->
                init_state_template d id;
                if(!needs_sync) then
                  (Buffer.add_string !compiled "wg.Add(1)\n ";);
                Buffer.add_string !compiled "go func(){\n";
                if(!needs_sync) then
                  (Buffer.add_string !compiled "defer wg.Done()\n ";);
                compile_exp type_map e env is_rec;
                Printf.bprintf !compiled "(%s, " d;
                compile_var_list_to_fun_args unr_args id_map;
                Buffer.add_string !compiled ")\n}()\n";
```

```
30                compile_c_proc type_map p id_map env is_rec
31            | (false, true) ->
32                init_state_template d id;
33                if(!needs_sync) then
34                  (Buffer.add_string !compiled "wg.Add(1)\n ";);
35                Buffer.add_string !compiled "go func(){\n";
36                if(!needs_sync) then
37                  (Buffer.add_string !compiled "defer wg.Done()\n ";);
38                compile_exp type_map e env is_rec;
39                Printf.bprintf !compiled "(%s, " d;
40                compile_var_list_to_fun_args lin_args id_map;
41                Buffer.add_string !compiled ")\n}()\n";
42                compile_c_proc type_map p id_map env is_rec
43            | (false, false) ->
44                init_state_template d id;
45                if(!needs_sync) then
46                  (Buffer.add_string !compiled "wg.Add(1)\n ";);
47                Buffer.add_string !compiled "go func(){\n";
48                if(!needs_sync) then
49                  (Buffer.add_string !compiled "defer wg.Done()\n ";);
50                compile_exp type_map e env is_rec;
51                Printf.bprintf !compiled "(%s, " d;
52                compile_var_list_to_fun_args lin_args id_map;
53                compile_var_list_to_fun_args unr_args id_map;
54                Buffer.add_string !compiled ")\n}()\n";
55                compile_c_proc type_map p id_map env is_rec
56            end
57          | None -> assert false
58          end
59      | None -> assert false
60      end
```

Care is taken to first obtain the actual state identifier in the current scope matching the process's channel name, then the next identifier in sequence is obtained, with an updated identifier map. When compiling a process, each individual action, such as sending or receiving a message, or spawning a new process, is not treated in isolation. The compiler also generates code for the continuation of the process, and the resulting Go code sequences both the current action and the subsequent operations, maintaining the precise order dictated by the session protocol.

If the compiled program makes use of cell states, the compiler introduces additional synchronization code to coordinate goroutines using a *WaitGroup*. In this case,

a `WaitGroup.Add(1)` statement is emitted before launching each goroutine in the main expression, while a **`defer`** `WaitGroup.Done()` call is inserted at the start of the corresponding goroutine body (as evidenced in the example above). Finally, a `WaitGroup.Wait()` invocation is appended to the end of the compiled main expression. This mechanism ensures that all goroutines complete their execution before the program terminates. The rationale and correctness of this coordination scheme are discussed in more detail later in this chapter. The two-phase approach implemented in the compiler is what ensures that the generated Go code faithfully represents the session-typed protocol while exploiting Go's concurrency primitives.

In the following sections we will follow, in some more detail, the compilation process of the most significant classical processes.

4.2.1 Servers

In this section, we follow the compilation process for the *Accept* and *Call* constructs, which begins with the creation of a server state. As with other states, this state is generated from a pre-built template and subsequently completed with additional information:

```
1   let make_from_srvr_template_single_channel name ty next =
2   Printf.bprintf !compiled "type %s struct {\n" name;
3   Printf.bprintf !compiled "   c chan interface{}\n";
4   Printf.bprintf !compiled "   next *%s\n}\n\n" next;
5   Printf.bprintf !compiled "func init%s(c chan interface{}) *%s {
6                                   return &%s{c, nil}
7                                 }\n"
8     name name name;
9   Printf.bprintf !compiled "func (x *%s) Call(v %s) {\n" name ty ;
10  Printf.bprintf !compiled
11    "  if x.next == nil { x.next = init%s(x.c) }; x.c <- v }\n" next;
12  Printf.bprintf !compiled "func (x *%s) Accept() (%s) {\n" name ty;
13  Printf.bprintf !compiled
14    "   if x.next == nil { x.next = init%s(x.c) };
15        return (<-x.c).(%s) }\n\n "
16    next ty
```

The following outlines the procedure for state creation as implemented in *make_state*:

```
1  let rec make_state map env stype =
2    match get_state map stype with
3    | STAffine (st, deps) | STCoAffine (st, deps) ->
4           let state_id = TypeStore.get_id stype in
5             let next_map, deps_list = make_deps_list map env deps in
```

```
6             begin match stype with
7             | STRec (v, _) ->
8                 begin match List.assoc_opt v env with
9                 | Some dummy_ref ->
10                    let this_state = State(state_id,
11                      Aff(TClassProc(st, [], []), dummy_ref, deps_list)
12                    ) in
13                    TypeTable.replace next_map stype this_state;
14                    next_map
15                | None ->
16                  let new_map = make_state next_map env st in
17                    let this_state =
18                      State (state_id, Aff(TClassProc(st, [], []),
19                        ref (TypeTable.find new_map st), deps_list))
20                    in
21                      TypeTable.replace new_map stype this_state;
22                      new_map
23                end
24            | _ ->
25              let new_map = make_state next_map env st in
26                let this_state =
27                  State (state_id, Aff(TClassProc(st, [], []),
28                    ref (TypeTable.find new_map st), deps_list))
29                in
30                  TypeTable.replace new_map stype this_state;
31                  new_map
32            end
```

As was discussed previously and now clearly shown, the state is implemented by a struct which contains a channel of type *interface*, and a reference to a state, which in the case of a server state is itself. Then we have the state initialization method, which is used lazily, whenever the previous state transitions to this one. In this method we receive the channel over which the communication will occur, as explained before this is needed because go channels cannot dynamically change their type in the same scope.

Then we have the *Call* and *Accept* methods which allow for the creation of a pipeline to wrap around the server's channel allowing the value passed through the channel to be passed along without a reference to the channel itself. It's also here we lazily initialize the *next* state, if not yet initialized. The parameters passed to this state are supplied in the *compile_state* function, which for the case of a server state the *name* and the *next* are both equal to the states's sequentially generated id, of the form ${}_s state_X$, where X represents the next numerical id in the sequence.

After preamble generation, the *compile_c_proc* compiles the *Accept* process as the following sequence of actions:

```
| Accept (id, d, opt, proc) ->
  let curr_d = curr_seq_id d id_map in
    let curr_id = curr_seq_id id id_map in
      let next_id, final_map = next_seq_id id id_map in
        begin match opt with
        | Some st ->
          begin match get_state type_map st with
          | Some (State(state_id, _)) ->
            Buffer.add_string !compiled "for {\n";
              Printf.bprintf !compiled "%s := %s.Accept()\n"
                curr_id curr_d;
              Printf.bprintf !compiled "%s := %s.(*%s)\n" next_id
                curr_id state_id;
              Printf.bprintf !compiled "go func(%s *%s) {\n" next_id
                state_id;
              compile_c_proc type_map proc final_map env is_rec;
              Printf.bprintf !compiled "}(%s)\n" next_id;
            Buffer.add_string !compiled "}\n"
          | None -> assert false
          end
        | None -> assert false
        end
```

Meaning that we first resolve the current runtime identifiers for the channels involved by using the sequential identifier map (*id_map*). This mechanism ensures that channels are consistently versioned, in order to respect the aforementioned channel usage constraints. The compiler then retrieves the state corresponding to the session type *st* from the *type_map*, which stores the association between session types and the runtime state structures generated for them.

Next, the compiler generates code that implements a server loop using an infinite *for* loop that repeatedly invokes the *Accept()* method on the shared endpoint to obtain a new connection. The returned value is dynamically cast to the state actually representing the server's body, identified by *state_id*, thus linking the runtime representation with the previously obtained session type. For every accepted connection, the compiler spawns a new goroutine that receives both the indetifier of the state representing the server's body and the appropriate version of the *id* channel as parameters. The server body's code is generated through recursive compilation of the continuation process *proc*. The updated identifier map is propagated through these compilation steps so that any subsequent

communication steps operate on the correctly versioned channel identifiers, maintaining the intended session progression.

And the *Call* as the following sequence of statements:

```
| Call (d, nc, opt, proc) ->
  let curr_id = curr_seq_id d id_map in
    let curr_nc = curr_seq_id nc id_map in
      begin match opt with
      | Some st ->
        begin match get_state type_map st with
        | Some (State(state_id, _)) ->
          init_state_template curr_nc state_id;
          Printf.bprintf !compiled "%s.Call(%s)\n" curr_id curr_nc;
          compile_c_proc type_map proc id_map env is_rec
        | None -> assert false
        end
      | None -> assert false
      end
```

Similarly to before, the compiler starts by resolving the current runtime identifiers for the process channels. It then retrieves the state corresponding to the session type of the continuation process from the *type_map*. This state represents the runtime structure associated with the protocol that will govern the interaction between the client and the server. Once the appropriate state structure is obtained, the compiler invokes *init_state_template* to generate the initialization code for the channel *nc*, instantiating it with the state identifier *state_id*. This step prepares the endpoint so that it conforms to the expected session protocol before the call is performed. Subsequently, the compiler emits code that invokes the *Call* method on the shared endpoint *d*, passing the initialized channel *nc* as argument. This operation establishes the session between the client and the server. Finally, the continuation process *proc* is compiled recursively as explained before.

4.2.2 Affine State

We will now examine the compilation of the *Affine*, *Use*, and *Discard* constructs, which are designed to manage channels representing affine processes. These constructs provide a mechanism to synchronize with and consume a decision made by another process, enabling safe, linear usage of resources within the protocol.

```
let make_from_affine_template_single_channel name next deps =
Printf.bprintf !compiled "type %s struct {\n" name;
Printf.bprintf !compiled "    c chan interface{}\n";
Printf.bprintf !compiled "    next *%s\n" next;
```

```
5   Printf.bprintf !compiled "    deps map[string]interface{ Dispose() }\n";
6   Printf.bprintf !compiled "    isDisposed bool\n";
7   Printf.bprintf !compiled "}\n";
8   Printf.bprintf !compiled "func init%s(c chan interface{}) *%s {\n"
9   name name;
10  Printf.bprintf !compiled "    deps := make(map[string]interface{
11                                        Dispose()
12                                        })\n";
13    make_from_deps_single_channel deps name;
14  Printf.bprintf !compiled "    return &%s{ c, nil, deps, false }\n" name;
15  Printf.bprintf !compiled "}\n\n";
16  Printf.bprintf !compiled "func (x *%s) AwaitDecision() (*%s, bool) {\n"
17    name next;
18  Printf.bprintf !compiled "    if x.next == nil {
19                                  x.next = init%s(x.c)
20                                }\n"
21    next;
22  Printf.bprintf !compiled "    _, ok := <- x.c;\n";
23  Printf.bprintf !compiled "    return %s, ok\n"
24    (if name != next then "x.next" else "x");
25  Printf.bprintf !compiled "}\n\n";
26  Printf.bprintf !compiled "func (x *%s) Use() *%s {\n" name next;
27  Printf.bprintf !compiled "    if x.next == nil {
28                                  x.next = init%s(x.c)
29                                }\n"
30    next;
31  Printf.bprintf !compiled "    x.c <- \"use\";\n";
32  Printf.bprintf !compiled "    return %s\n"
33    (if name != next then "x.next" else "x");
34  Printf.bprintf !compiled "}\n\n";
35  Printf.bprintf !compiled "func (x *%s) Dispose() {\n" name;
36  Printf.bprintf !compiled "    if x.isDisposed {\n";
37  Printf.bprintf !compiled "        return\n";
38  Printf.bprintf !compiled "    }\n";
39  Printf.bprintf !compiled "    x.isDisposed = true\n";
40  Printf.bprintf !compiled "    for _, dep := range x.deps {\n";
41  Printf.bprintf !compiled "        dep.Dispose()\n";
42  Printf.bprintf !compiled "    }\n";
43  Printf.bprintf !compiled "    close(x.c)\n";
44  Printf.bprintf !compiled "}\n\n"
```

The affine state is created using the pre-defined template described above, while the *make_state* function executes the procedure below to instantiate the state:

```
| STAffine (st, deps) | STCoAffine (st, deps) ->
        let state_id = TypeStore.get_id stype in
          let next_map, deps_list = make_deps_list map env deps in
          begin match stype with
          | STRec (v, _) ->
              begin match List.assoc_opt v env with
              | Some dummy_ref ->
                  let this_state = State(state_id,
                    Aff(TClassProc(st, [], []), dummy_ref, deps_list)) in
                  TypeTable.replace next_map stype this_state;
                  next_map
              | None ->
                let new_map = make_state next_map env st in
                  let this_state =
                    State (state_id, Aff(TClassProc(st, [], []),
                      ref (TypeTable.find new_map st), deps_list))
                  in
                    TypeTable.replace new_map stype this_state;
                    new_map
              end
          | _ ->
            let new_map = make_state next_map env st in
              let this_state =
                State (state_id, Aff(TClassProc(st, [], []),
                  ref (TypeTable.find new_map st), deps_list))
              in
                TypeTable.replace new_map stype this_state;
                new_map
          end
```

First, the current identifier for d is resolved using *id_map*, ensuring correct channel versioning. The compiler then emits code that calls *AwaitDecision()* on the channel, returning both the next channel version and a Boolean flag (*ok_flag*) indicating whether the resource should be used or discarded. The function executing this code blocks until the decision is received.

Next, the compiler enables the interpretation of the decision by emitting a conditional that terminates the process if the channel was closed with a discard (meaning the *ok_flag* is false), preventing further operations on an invalid resource. The continuation process

proc is then recursively compiled using *compile_c_proc* with the updated identifier map. If the decision instead indicates that the resource should be used, the conditional is skipped and execution continues normally, allowing subsequent communication steps to proceed in accordance with the protocol.

```
| Affine (d, _, _, proc) ->
    let curr_d = curr_seq_id d id_map in
      let next_d, final_map = next_seq_id d id_map in
        Printf.bprintf !compiled "%s, ok_flag := %s.AwaitDecision()\n"
            next_d curr_d;
        Printf.bprintf !compiled "if (!ok_flag) {\n return;\n}\n";
        compile_c_proc type_map proc final_map env is_rec
```

```
| Use (d, _, proc) ->
  let curr_d = curr_seq_id d id_map in
    let next_d, final_map = next_seq_id d id_map in
      Printf.bprintf !compiled "%s := %s.Use()\n" next_d curr_d;
      compile_c_proc type_map proc final_map env is_rec
```

After resolving the current channel identifier and determining the next sequential version, The compiler then emits a call to the *Use()* method on the channel, which sends the decision to use the affine resource through the channel and yields the next channel version. This effectively consumes the affine capability and prepares the endpoint for further linear interactions. As with *Affine*, the continuation process is recursively compiled using the updated identifier mapping.

```
| Discard (d, _, proc) ->
  Buffer.add_string !compiled (curr_seq_id d id_map);
  Buffer.add_string !compiled ".Dispose()\n";
  compile_c_proc type_map proc id_map env is_rec
```

The compiler obtains the correct identifier for the channel *d*, then emits a call to the *Dispose()* method on the channel, signaling that the affine resource is to be discarded. This operation closes the channel and disposes of any possible dependencies the affine state may possess. This prevents further usage and ensures that any linear capabilities associated with this state are safely released. The continuation process *proc* is then also recursively compiled using the existing identifier mapping, allowing subsequent steps in the protocol to proceed with correctly versioned channels.

At the end of this chapter we show an example of a compiled program using affine session inside shared mutable cells.

4.2.3 Shared Mutable Cells

Compiling mutually exclusive shared mutable cells is both a challenging and central component of our system. To accurately capture the runtime behavior, the cell state is constructed from the template shown below, augmented with a sequentially generated state identifier and the identifier for the state representing the stored process's session type.

```
1  let make_from_cell_template_single_channel name stored_state =
2  Printf.bprintf !compiled "type %s struct {\n" name;
3  Printf.bprintf !compiled "    c chan interface{}\n";
4  Printf.bprintf !compiled "    stored_state *%s\n" stored_state;
5  Printf.bprintf !compiled "    proc func(*%s)\n" stored_state;
6  Printf.bprintf !compiled "    state_lock *sync.Mutex\n";
7  Printf.bprintf !compiled "    run_lock *sync.Cond\n";
8  Printf.bprintf !compiled "    isRunning bool\n";
9  Printf.bprintf !compiled "    isDisposed bool\n";
10 Printf.bprintf !compiled "    scheduledOps int\n";
11 Printf.bprintf !compiled "    completedOps int\n";
12 Printf.bprintf !compiled "}\n";
13 Printf.bprintf !compiled "func init%s(c chan interface{}) *%s {\n"
14   name name;
15 Printf.bprintf !compiled "    state_lock := &sync.Mutex {}\n";
16 Printf.bprintf !compiled "    x := &%s{ c, nil, nil, state_lock,
17                                 sync.NewCond(state_lock),
18                                 false, false, 1, 0 }\n" name;
19 Printf.bprintf !compiled "    return x\n";
20 Printf.bprintf !compiled "}\n\n";
21 Printf.bprintf !compiled "func (x *%s) Release() {\n" name;
22 Printf.bprintf !compiled "x.state_lock.Lock()\n
23                           defer x.state_lock.Unlock()\n";
24 Printf.bprintf !compiled "x.completedOps++\n";
25 Printf.bprintf !compiled "if (x.completedOps == x.scheduledOps) {
26                             x.Dispose()
27                           } \n";
28 Printf.bprintf !compiled "}\n\n";
29 Printf.bprintf !compiled "func (x *%s) Dispose() {\n" name;
30 Printf.bprintf !compiled "    if x.isDisposed {\n";
31 Printf.bprintf !compiled "        return\n";
32 Printf.bprintf !compiled "    }\n";
33 Printf.bprintf !compiled "    x.isDisposed = true\n";
34 Printf.bprintf !compiled "    if x.proc != nil{\n
```

```
35                                  if x.stored_state == nil {
36                                    x.stored_state = init%s(x.c)
37                                  }\n
38                                  x.stored_state.Dispose() } \n" stored_state;
39    Printf.bprintf !compiled "}\n\n";
40    Printf.bprintf !compiled "func (x *%s) Take(a *%s) {\n" name stored_state;
41    Printf.bprintf !compiled "    x.state_lock.Lock()\n";
42    Printf.bprintf !compiled "    if(x.isDisposed){
43                                    x.state_lock.Unlock()\n return;
44                                  }\n";
45    Printf.bprintf !compiled "    for x.proc == nil
46                                      || x.isRunning { x.run_lock.Wait() }\n";
47    Printf.bprintf !compiled "    \n";
48    Printf.bprintf !compiled "    p := x.proc\n";
49    Printf.bprintf !compiled "    x.proc = nil\n";
50    Printf.bprintf !compiled "    x.isRunning = true\n";
51    Printf.bprintf !compiled "    x.state_lock.Unlock()\n";
52    Printf.bprintf !compiled "    go func(){\n";
53    Printf.bprintf !compiled "    p(a)\n";
54    Printf.bprintf !compiled "    x.state_lock.Lock()\n";
55    Printf.bprintf !compiled "    x.isRunning = false\n";
56    Printf.bprintf !compiled "    x.run_lock.Broadcast()\n";
57    Printf.bprintf !compiled "    x.state_lock.Unlock()\n";
58    Printf.bprintf !compiled "    }()\n\n";
59    Printf.bprintf !compiled "}\n\n";
60    Printf.bprintf !compiled "func (x *%s) Put(new_proc func(*%s)) {\n"
61      name stored_state;
62    Printf.bprintf !compiled "    x.state_lock.Lock()\n";
63    Printf.bprintf !compiled "    if(x.isDisposed){
64                                    x.state_lock.Unlock()\n return;
65                                  }\n";
66    Printf.bprintf !compiled "    for x.proc != nil || x.isRunning {
67                                    x.run_lock.Wait()
68                                  }\n";
69    Printf.bprintf !compiled "    \n";
70    Printf.bprintf !compiled "    x.proc = new_proc\n";
71    Printf.bprintf !compiled "    x.run_lock.Broadcast()\n";
72    Printf.bprintf !compiled "    x.state_lock.Unlock()\n";
73    Printf.bprintf !compiled "}\n\n"
```

The cell state is produced by the *make_state* function according to the procedure shown

below. Note that all session types corresponding to cell processes compile to the same state, meaning it must represent all the behaviors needed during execution.

```
  | STFullState (stored_st) | STFullUse (stored_st)
      | STEmptyState (stored_st) | STEmptyUse (stored_st) ->
        needs_sync := true;
        let state_id = TypeStore.get_id (STFullState(stored_st)) in
          let affine_st = STAffine(stored_st, []) in
            let new_map = make_state map env affine_st in
              let this_state =
                State(state_id, Cell(TClassProc(affine_st, [], []),
                  ref (TypeTable.find new_map affine_st))) in
                TypeTable.replace new_map (STFullState(stored_st)) this_state;
                new_map
```

The compilation of a *Cell* construct begins by resolving the current identifiers for both the target channel d and the associated affine resource a. The compiler then generates a new function that wraps the continuation process q, instantiating it with the stored state corresponding to the affine resource. This function is subsequently passed to the channel d using its *Put()* method, effectively generating a full cell. By encapsulating the process in a function, the compiler ensures that the affine resource is correctly isolated and that its linear usage semantics are preserved.

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Cell (d, opt, a, q) ->
    let curr_d = curr_seq_id d id_map in
    let curr_a = curr_seq_id a id_map in
      let (func_name, final_map) = gen_seq_id "new_func" id_map in
        begin match opt with
        | Some st ->
          begin match get_state type_map (STAffine(st, [])) with
          | Some (State(stored_state_id, _)) ->
            Printf.bprintf !compiled "//Compiled Cell\n";
            Printf.bprintf !compiled "%s := func(%s *%s) {\n"
              func_name curr_a stored_state_id;
            compile_c_proc type_map q final_map env is_rec;
            Printf.bprintf !compiled "}\n";
            Printf.bprintf !compiled "%s.Put(%s)\n" curr_d func_name;
          | None -> assert false
          end
```

```
        | None -> assert false
        end
```

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Empty _ ->
    Printf.bprintf !compiled "//Compiled Empty Cell\n";
```

The *Empty* construct is compiled as a no-op placeholder, indicating that the cell contains no active process or value at this point in execution.

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Take (d, opt, a, proc) ->
    let curr_d = curr_seq_id d id_map in
      let curr_a = curr_seq_id a id_map in
        begin match opt with
        | Some st ->
          begin match get_state type_map (STFullUse(st)) with
          | Some (State(state_id, _)) ->
            begin match get_state type_map (STCoAffine(st, [])) with
            | Some (State(stored_state_id, _))->
              init_state_template curr_a stored_state_id;
              Buffer.add_string !compiled "wg.Add(1)\n ";
              Printf.bprintf !compiled "//Started Compiling Take\n";
              Printf.bprintf !compiled "go func(%s *%s, %s *%s){\n"
                curr_d state_id curr_a stored_state_id;
                Buffer.add_string !compiled "defer wg.Done()\n ";
                Printf.bprintf !compiled "%s.Take(%s)\n" curr_d curr_a;
                compile_c_proc type_map proc id_map env is_rec;
              Printf.bprintf !compiled "}(%s, %s)\n\n" curr_d curr_a;
              Printf.bprintf !compiled "//Finished Compiling Take\n";
            | None -> assert false
            end
          | None -> assert false
          end
        | None -> assert false
        end
```

For the *Take* construct, the compiler resolves the identifiers for the channel d and the affine resource a, initializes the resource state if necessary, and emits code that invokes the *Take()* method. This method consumes the stored process in the cell and executes it in a

new goroutine, passing the affine resource as an argument. The continuation process *proc* is then recursively compiled within the same scope, ensuring that subsequent protocol steps operate on the correctly versioned channels. This approach enforces linearity while allowing concurrent execution of the cell's content.

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Put (d, opt, a, q, proc) ->
    let curr_d = curr_seq_id d id_map in
      let curr_a = curr_seq_id a id_map in
        let (func_name, final_map) = gen_seq_id "new_func" id_map in
        begin match opt with
        | Some st ->
              begin match get_state type_map (STCoAffine(st, [])) with
              | Some (State(stored_state_id, _)) ->
                  Printf.bprintf !compiled "%s := func(%s *%s) {\n"
                    func_name curr_a stored_state_id;
                  compile_c_proc type_map q final_map env is_rec;
                  Printf.bprintf !compiled "}\n";
                    Printf.bprintf !compiled "%s.Put(%s)\n" curr_d
                      func_name;
                  compile_c_proc type_map proc final_map env is_rec
              | None -> assert false
              end
        | None -> assert false
        end
```

The compilation of a *Put* construct follows a complementary pattern. The compiler first resolves the identifiers for the channel d and the affine resource a, then generates a new function encapsulating the continuation process q. This function is delivered to the cell via the *Put()* method on d, scheduling it for later execution. Afterward, the continuation process *proc* is recursively compiled with the updated identifier map, propagating the correctly versioned channels to subsequent steps in the protocol.

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Release (d, proc) ->
    let curr_d = curr_seq_id d id_map in
      Printf.bprintf !compiled "%s.Release()\n" curr_d;
      compile_c_proc type_map proc id_map env is_rec
```

The *Release* construct is compiled by emitting a call to the *Release()* method on the channel. This operation decrements the scheduled operation count and may trigger

disposal of the cell if all scheduled operations have completed. By doing so, it ensures that affine resources are properly released and that no lingering references violate linear usage constraints. The continuation process is then recursively compiled using the existing identifier mapping.

Finally, the *Share* construct allows the creation of concurrent subscriptions to a shared mutable cell modelled by the cell state. The compiler resolves the identifier for the channel d and retrieves the corresponding state. It emits code that increments the scheduled operations counter and launches a new goroutine executing the shared process q. The continuation process r is then recursively compiled, ensuring that both the shared execution and subsequent protocol steps maintain correct channel versioning and linearity.

```
let rec compile_c_proc type_map p id_map env is_rec =
  match p with
  | Share (d, opt, q, r) ->
    let curr_d = curr_seq_id d id_map in
      begin match opt with
      | Some st ->
        begin match get_state type_map st with
        | Some (State(state_id, _)) ->
          Buffer.add_string !compiled "wg.Add(1)\n ";
          Printf.bprintf !compiled "%s.scheduledOps++\n " curr_d;
          Printf.bprintf !compiled "go func(%s *%s){\n" curr_d state_id;
            Buffer.add_string !compiled "defer wg.Done()\n ";
            compile_c_proc type_map q id_map env is_rec;
          Printf.bprintf !compiled "}(%s)\n" curr_d;
          compile_c_proc type_map r id_map env is_rec;
        | None -> assert false
        end
      | None -> assert false
      end
```

At runtime, interactions with a shared memory cell follow a disciplined pattern that respects the linearity of its stored process. A cell may initially be empty or full, with a full cell containing a process associated with an affine session, meaning the process can either be used or discarded. When a cell is full, invoking *Take()* consumes the stored process and launches it in a separate goroutine, temporarily locking the cell to prevent other concurrent processes from interfering. The process that executed the *Take()* then proceeds with execution according to the compiled continuation process. This continuation will interact with the process in the goroutine, after which the cell can be repopulated via *Put()*, unlocking it for further interactions. If the cell is empty, it must first be populated with a process using *Put()*, before the protocol can proceed with execution. Whenever a cell

is full, the interacting processes can release their access to the cell by invoking *Release()*. When all processes release, the cell is discarded along with the process it is storing.

The *Share()* construct allows multiple subscribers to access the cell concurrently. Depending on the cell's state, a subscriber may either perform a *Take()–Put()* cycle or populate an empty cell, ensuring that all interactions conform to the linear and affine guarantees imposed by the session type.

Below is an example of a compiled program from our language. This particular example is the compilation result of the first shared memory cell example, from Section 3.3:

```
  package main
import "fmt"
import "sync"
//Preamble generation
type _state_4 struct {
    c chan interface{}
}
func init_state_4(c chan interface{}) *_state_4 { return &_state_4{ c } }
func (x *_state_4) Send(v interface{}) { x.c <- v }
func (x *_state_4) Recv() interface{} { return <-x.c }

  type _state_3 struct {
   c chan interface{}
   next *_state_4
}

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

  type _state_2 struct {
   c chan interface{}
   next *_state_3
}

func init_state_2(c chan interface{}) *_state_2 { return &_state_2{c, nil} }
```

```
33 func (x *_state_2) Send(v int) *_state_3 {
34    if x.next == nil {
35     x.next = init_state_3(x.c) }; x.c <- v; return x.next
36   }
37 func (x *_state_2) Recv() (int, *_state_3) {
38    if x.next == nil {
39       x.next = init_state_3(x.c) }; return (<-x.c).(int), x.next
40     }
41
42   type _state_1 struct {
43     c chan interface{}
44     next *_state_2
45     deps map[string]interface{ Dispose() }
46     isDisposed bool
47 }
48 func init_state_1(c chan interface{}) *_state_1 {
49     deps := make(map[string]interface{ Dispose() })
50     return &_state_1{ c, nil, deps, false }
51 }
52
53 func (x *_state_1) AwaitDecision() (*_state_2, bool) {
54     if x.next == nil { x.next = init_state_2(x.c) }
55     _, ok := <- x.c;
56     return x.next, ok
57 }
58
59 func (x *_state_1) Use() *_state_2 {
60     if x.next == nil { x.next = init_state_2(x.c) }
61     x.c <- "use";
62     return x.next
63 }
64
65 func (x *_state_1) Dispose() {
66     if x.isDisposed {
67         return
68     }
69     x.isDisposed = true
70     for _, dep := range x.deps {
71         dep.Dispose()
72     }
73     close(x.c)
```

```
74  }
75
76  type _state_0 struct {
77      c chan interface{}
78      stored_state *_state_1
79      proc func(*_state_1)
80      state_lock *sync.Mutex
81      run_lock *sync.Cond
82      isRunning bool
83      isDisposed bool
84      scheduledOps int
85      completedOps int
86  }
87  func init_state_0(c chan interface{}) *_state_0 {
88      state_lock := &sync.Mutex {}
89      x := &_state_0{ c, nil, nil, state_lock,
90        sync.NewCond(state_lock), false, false, 1, 0 }
91      return x
92  }
93
94  func (x *_state_0) Release() {
95  x.state_lock.Lock()
96   defer x.state_lock.Unlock()
97  x.completedOps++
98  if (x.completedOps == x.scheduledOps) { x.Dispose() }
99  }
100
101 func (x *_state_0) Dispose() {
102     if x.isDisposed {
103         return
104     }
105     x.isDisposed = true
106     if x.proc != nil{
107  if x.stored_state == nil { x.stored_state = init_state_1(x.c) }
108  x.stored_state.Dispose() }
109 }
110
111 func (x *_state_0) Take(a *_state_1) {
112     x.state_lock.Lock()
113     if(x.isDisposed){ x.state_lock.Unlock()
114  return; }
```

```
115     for x.proc == nil || x.isRunning { x.run_lock.Wait() }
116
117     p := x.proc
118     x.proc = nil
119     x.isRunning = true
120     x.state_lock.Unlock()
121     go func(){
122     p(a)
123     x.state_lock.Lock()
124     x.isRunning = false
125     x.run_lock.Broadcast()
126     x.state_lock.Unlock()
127     }()
128
129 }
130
131 func (x *_state_0) Put(new_proc func(*_state_1)) {
132     x.state_lock.Lock()
133     if(x.isDisposed){ x.state_lock.Unlock()
134  return; }
135     for x.proc != nil || x.isRunning { x.run_lock.Wait() }
136
137     x.proc = new_proc
138     x.run_lock.Broadcast()
139     x.state_lock.Unlock()
140 }
141
142 //Declaration list compilation
143 func server()func (_x *_state_0) {
144  return func (f *_state_0){
145 //Compiled Cell
146 new_func0 := func(p *_state_1) {
147 p0, ok_flag := p.AwaitDecision()
148 //If the channel was closed with a discard,
149 //  we must end the execution gracefully
150 if (!ok_flag) {
151  return;
152 }
153 x, p1 := p0.Recv()
154 p2 := p1.Send(4)
155 fmt.Printf("%v\n",x)
```

```
156 p2.Send(nil)
157 }
158 f.Put(new_func0)
159 }}
160 //Main compilation
161 func main () {
162 m := init_state_4(make(chan interface{}))
163 var wg sync.WaitGroup
164 wg.Add(1)
165 go func () {
166 defer wg.Done()
167 m.Recv()
168 }()
169 func (m *_state_4){
170 t := init_state_0(make(chan interface{}))
171 wg.Add(1)
172  go func(){
173  defer wg.Done()
174  server()(t)
175 }()
176 wg.Add(1)
177  t.scheduledOps++
178  go func(t *_state_0){
179 defer wg.Done()
180  fmt.Printf("%v\n",10)
181 a := init_state_1(make(chan interface{}))
182 wg.Add(1)
183  //Started Compiling Take
184 go func(t *_state_0, a *_state_1){
185 defer wg.Done()
186  t.Take(a)
187 a0 := a.Use()
188 a1 := a0.Send(4)
189 x, a2 := a1.Recv()
190 fmt.Printf("%v\n",x)
191 a2.Recv()
192 new_func0 := func(a2 *_state_1) {
193 a3, ok_flag := a2.AwaitDecision()
194 //If the channel was closed with a discard, we must end the execution gracefully
195 if (!ok_flag) {
196  return;
```

```
197 }
198 _, a4 := a3.Recv()
199 a5 := a4.Send(4)
200 a5.Send(nil)
201 }
202 t.Put(new_func0)
203 t.Release()
204 //Terminated process
205 }(t, a)
206
207 //Finished Compiling Take
208 }(t)
209 wg.Add(1)
210  t.scheduledOps++
211  go func(t *_state_0){
212 defer wg.Done()
213  fmt.Printf("%v\n",20)
214 a := init_state_1(make(chan interface{}))
215 wg.Add(1)
216  //Started Compiling Take
217 go func(t *_state_0, a *_state_1){
218 defer wg.Done()
219  t.Take(a)
220 a0 := a.Use()
221 a1 := a0.Send(2)
222 x, a2 := a1.Recv()
223 a2.Recv()
224 fmt.Printf("%v\n",x)
225 new_func0 := func(a2 *_state_1) {
226 a3, ok_flag := a2.AwaitDecision()
227 //If the channel was closed with a discard,
228 //  we must end the execution gracefully
229 if (!ok_flag) {
230  return;
231 }
232 _, a4 := a3.Recv()
233 a5 := a4.Send(4)
234 a5.Send(nil)
235 }
236 t.Put(new_func0)
237 t.Release()
```

```
238 m.Send(nil)
239 }(t, a)
240
241 //Finished Compiling Take
242 }(t)
243 fmt.Printf("%v\n",30)
244 t.Release()
245 //Terminated process
246 }(m)
247 wg.Wait()
248 }
```

5

Performance Evaluation

Performance evaluation is essential in exploratory work, as it enables meaningful conclusions to be drawn about the system under study. In this case, it helps us better understand the limitations of the proposed language and identify potential directions for future improvements. For this purpose, we conducted a performance analysis focusing on both compilation and execution times.

Our primary goal is to determine how our additions affected the system originally designed by J. Geraldo. Consequently, our evaluation focuses on comparing compilation times between the two Go channel management approaches briefly described in Chapter 4, Section 4.2.

To recall, J. Geraldo [14] previously analyzed the performance of two executions of the same program compiled using different approaches: the *single-channel* approach and the *multi-channel* approach. His results indicated that the difference between them is marginal, as the overhead introduced by creating multiple channels outweighs any potential overhead caused by the typecasting process.

Below we present the test program we decide to use for our performance evaluation, note that it is a direct translation from the program used by Geraldo, so that our results are comparable.

```
1  package main
2  //Preamble not shown for presentation purposes
3
4  func main () {
5  c := init_state_0(make(chan interface{}))
6  go func () {
7  c.Recv()
8  }()
9  func (c *_state_0){
10 d := init_state_1(make(chan interface{}))
11 go func(){
12  func (d *_state_1){
```

```
13
14 v1, d0 := d.Recv();
15 v2, d1 := d0.Recv(); v3, d2 := d1.Recv(); v4, d3 := d2.Recv();
16 v5, d4 := d3.Recv(); v6, d5 := d4.Recv(); v7, d6 := d5.Recv();
17 v8, d7 := d6.Recv(); v9, d8 := d7.Recv(); v10, d9 := d8.Recv();
18
19 d10 := d9.Send(v1)
20 d11 := d10.Send(v2); d12 := d11.Send(v3); d13 := d12.Send(v4);
21 d14 := d13.Send(v5); d15 := d14.Send(v6); d16 := d15.Send(v7);
22 d17 := d16.Send(v8); d18 := d17.Send(v9); d19 := d18.Send(v10);
23
24 d19.Send(nil)
25 }(d)
26 }()
27 d0 := d.Send(1);
28 d1 := d0.Send(2); d2 := d1.Send(3); d3 := d2.Send(4);
29 d4 := d3.Send(5); d5 := d4.Send(6); d6 := d5.Send(7);
30 d7 := d6.Send(8); d8 := d7.Send(9); d9 := d8.Send(10);
31
32
33 _, d10 := d9.Recv();
34 _, d11 := d10.Recv(); _, d12 := d11.Recv(); _, d13 := d12.Recv();
35 _, d14 := d13.Recv(); _, d15 := d14.Recv(); _, d16 := d15.Recv();
36 _, d17 := d16.Recv(); _, d18 := d17.Recv(); _, d19 := d18.Recv();
37
38 d19.Recv()
39 c.Send(nil)
40 }(c)
41 }
42
```

5.1 Results

To evaluate the performance of code generated by our compiler, we executed the previously compiled test program 100 times, recording the execution time for each iteration. We then calculated the mean and standard deviation, after removing outliers. These outliers were likely caused by fluctuations in the performance of the host machine rather than the compiled code itself, so they were excluded from the statistical analysis.

The results are presented below:

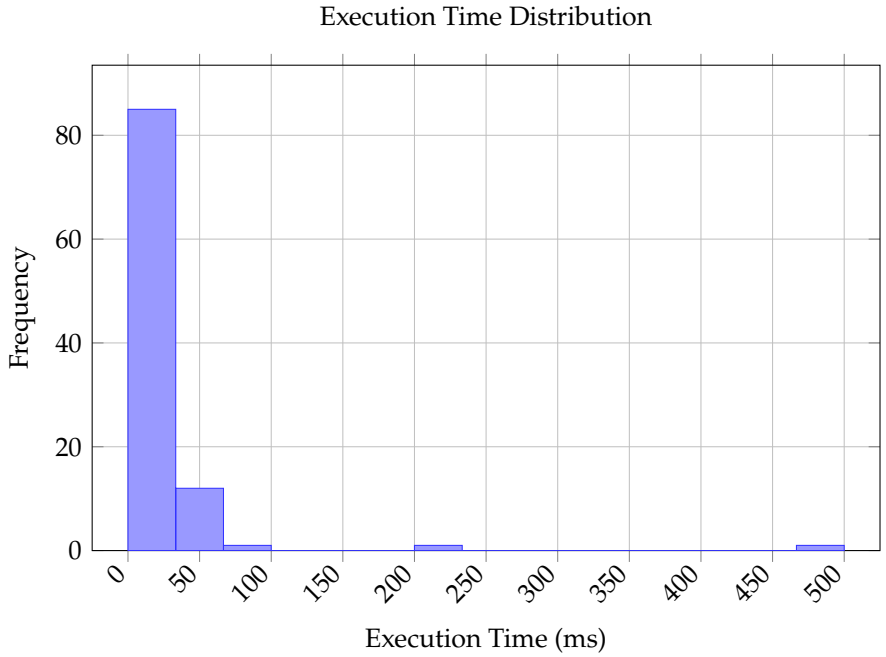

Figure 5.1: Distribution of execution times across runs (Single-Channel).

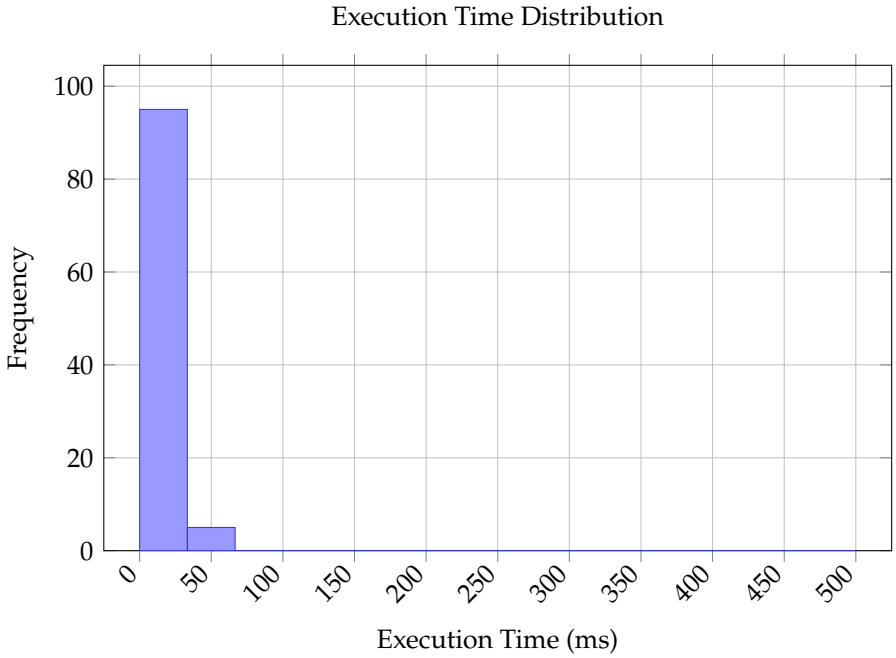

Figure 5.2: Distribution of execution times across runs (Multi-Channel).

These results (and the ones below) indicate that the multi-channel approach outperforms the alternatives, contrary to the findings reported by Geraldo (presented below). Additionally, our extension appears to slightly increase execution times relative to Geraldo's results. This difference is small and may be attributable to variability in machine

performance rather than the changes introduced by our extension.

Table 5.1: Execution Times Statistic in Our Language

	Mean Value	Standard Deviation
Single-Channel	23.643	13.147
Multi-Channel	18.230	7.990

Table 5.2: Average Execution Times in Geraldo's Language in Microseconds

	Mean Value
Single-Channel	18.925
Multi-Channel	17.424

We also measured compilation times for both channel approaches by compiling the same test program approximately 200 times and recording the compilation time for each iteration. The resulting average times are presented below:

Table 5.3: Average Compilation Times in Our Language in Microseconds

	Mean Value
Single-Channel	614.28
Multi-Channel	808.92

These results indicate that the multi-channel approach compiles more slowly than the single-channel approach. This is likely due to the additional code required to manage multiple channels, including synchronization and data transfer.

6

Conclusion

In this work, we extended existing systems to develop a simple interpreter and a full-featured typechecker. By leveraging bidirectional typechecking, our type system ensures correctness while allowing most type annotations to be omitted. We also implemented a compiler that translates programs written in our language into equivalent, executable Go code, using specialized encoding strategies to map session types directly onto Go constructs. Finally, we conducted a performance evaluation of the enhanced language, comparing it with the previous implementation to highlight its limitations and identify opportunities for further improvement.

This language provides compile-time guarantees that well-typed programs are free from communication errors and deadlocks.

For future work, several directions are particularly promising. One is to broaden the set of programs accepted by the typechecker, for example by implementing polymorphic session types as described in CLASS [22], which would substantially increase the language's expressiveness. Other avenues include optimizing the compilation process, building on the improvements J. Faria implemented in J. Geraldo's work [13], or generalizing the *fwd* construct to support forwarding of shared memory cells and affine state along with their dependencies.

Bibliography

[1] S. Abramsky. "Computational interpretations of linear logic". In: *Theoretical computer science* 111.1-2 (1993), pp. 3–57 (cit. on p. 2).

[2] S. Abramsky and A. Jung. "Domain theory". In: (1994) (cit. on p. 2).

[3] B. Almeida, A. Mordido, and V. T. Vasconcelos. "FreeST: Context-free Session Types in a Functional Language". In: *Electronic Proceedings in Theoretical Computer Science* 291 (2019-04), pp. 12–23. ISSN: 2075-2180. DOI: 10.4204/eptcs.291.2. URL: http://dx.doi.org/10.4204/EPTCS.291.2 (cit. on p. 22).

[4] B. Almeida, A. Mordido, and V. T. Vasconcelos. "FreeST: Context-free session types in a functional language". In: *arXiv preprint arXiv:1904.01284* (2019) (cit. on p. 17).

[5] E. Beffara. "Introduction to linear logic". In: (2013) (cit. on p. 5).

[6] G. Bellin and P. J. Scott. "On the π-calculus and linear logic". In: *Theoretical Computer Science* 135.1 (1994), pp. 11–65 (cit. on p. 2).

[7] T. Braüner. *Introduction to linear logic*. Computer Science Department, 1996 (cit. on pp. 5, 7).

[8] L. Caires and F. Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory*. Ed. by P. Gastin and F. Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 222–236. ISBN: 978-3-642-15375-4 (cit. on pp. 13–15).

[9] L. CAIRES, F. PFENNING, and B. TONINHO. "Linear logic propositions as session types". In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423. DOI: 10.1017/S0960129514000218 (cit. on pp. 2, 8).

[10] L. Caires et al. "Behavioral polymorphism and parametricity in session-based communication". In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, pp. 330–349 (cit. on p. 18).

[11] I. Cervesato, J. S. Hodas, and F. Pfenning. "Efficient resource management for linear logic proof search". In: *Theoretical Computer Science* 232.1-2 (2000), pp. 133–163 (cit. on p. 45).

[12] T. Ehrhard. "An introduction to differential linear logic: proof-nets, models and antiderivatives". In: *Mathematical Structures in Computer Science* 28.7 (2018), pp. 995–1060 (cit. on p. 21).

[13] J. Faria. "An optimizing compiler for a session-typed functional language". PhD thesis. Master Thesis in Computer Science. NOVA School of Science and Technology, 2023 (cit. on p. 113).

[14] J. Geraldo. "Making Session Types Go". MA thesis. NOVA University Lisbon - NOVA School of Science and Technology, 2022 (cit. on pp. v, vi, 2, 3, 22, 23, 27, 31, 44, 45, 53, 80, 109).

[15] J.-Y. Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. URL: https://www.sciencedirect.com/science/article/pii/0304397587900454 (cit. on pp. 4, 5).

[16] C. A. R. Hoare et al. *Communicating sequential processes*. Vol. 178. Prentice-hall Englewood Cliffs, 1985 (cit. on p. 9).

[17] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: *European Symposium on Programming*. Springer. 1998, pp. 122–138 (cit. on p. 9).

[18] W. Howard. "Jean-Yves Girard, Paul Taylor, and Yves LaFont. Proofs and types. Cambridge tracts in theoretical computer science, no. 7. Cambridge University Press, Cambridge etc. 1989, xi+ 176 pp." In: *The Journal of Symbolic Logic* 56.2 (1991), pp. 760–761 (cit. on p. 19).

[19] H. Hüttel et al. "Foundations of session types and behavioural contracts". In: *ACM Computing Surveys (CSUR)* 49.1 (2016), pp. 1–36 (cit. on p. 9).

[20] 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).

[21] P. Rocha and L. Caires. "Propositions-as-types and shared state". In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–30 (cit. on pp. 13, 27).

[22] P. Rocha and L. Caires. "Safe session-based concurrency with shared linear state". In: *European Symposium on Programming*. Springer Nature Switzerland Cham. 2023, pp. 421–450 (cit. on pp. 2, 13, 17, 23, 27, 31, 34, 113).

[23] P. Thiemann and V. T. Vasconcelos. "Context-free session types". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 462–475 (cit. on p. 17).

[24] V. T. Vasconcelos. "Fundamentals of session types". In: *Information and Computation* 217 (2012), pp. 52–70 (cit. on p. 11).

[25] P. Wadler. "Propositions as sessions". In: *SIGPLAN Not.* 47.9 (2012-09), pp. 273–286. ISSN: 0362-1340. DOI: 10.1145/2398856.2364568. URL: https://doi.org/10.1145/2398856.2364568 (cit. on pp. 2, 13).

[26] M. Willsey, R. Prabhu, and F. Pfenning. "Design and Implementation of Concurrent C0". In: *Electronic Proceedings in Theoretical Computer Science* 238 (2017-01), pp. 73–82. ISSN: 2075-2180. DOI: 10.4204/eptcs.238.8. URL: http://dx.doi.org/10.4204/EPTCS.238.8 (cit. on p. 22).