DEPARTMENT OF
COMPUTER SCIENCE

JOÃO MIGUEL PEREIRA DO CANO RICO GERALDO
BSc in Computer Science and Engineering

MAKING SESSION TYPES GO

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2022

DEPARTMENT OF
COMPUTER SCIENCE

MAKING SESSION TYPES GO

JOÃO MIGUEL PEREIRA DO CANO RICO GERALDO
BSc in Computer Science and Engineering

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

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2022

Making Session Types Go

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.14) [26].

To you, who endured.

Acknowledgements

It’s been quite the journey, quite longer than I had been expecting at the start, but it’s finally over and there are a lot of people who helped me along the way and whom I would like to thank.

I’d like to start by thanking my advisor, Professor Bernardo Toninho, without whom I would have *literally* been unable to do this thesis. Unfailingly capable, and ready to help, I lost count of the number of times he had to give the same explanation for things I thought I understood, but not really. He was always patient and understanding, and he never made me feel dumb or bad for not understanding or failing to meet my goals. Truly more than just a good professor, or a good advisor, to me Professor Toninho is, more than anything, a great person. I only wish I could have been a better student, better at the work, to provide Professor Toninho with a more fulfilling advising opportunity. For that I am sorry. Nevertheless, I thank you, Professor Toninho, for all the help, from the bottom of my heart.

Next I’d like to thank my family. My mom Agostinha, my dad Eduardo, and my brother Didi. They always stood by me. They never once showed any disappointment in me for not finishing on time, even when I felt it so desperately myself. They showed me endless, unconditional, support, and their presence during the lonelier days of the pandemic kept me company, warming my heart, and allowing me to better deal with life’s various (sometimes apparently insurmountable) hurdles. It’s hard to imagine how things would have turned out if it were not for the three of them. So, for that I am thankful. To all of you. I also want to thank my brother’s girlfriend Joana, who not only rooted for me to reach my goal, but also contributed to me finishing this thesis, by providing me with some technical knowledge on statistics. Thanks Joana!

I’d also like to thank all my friends! They provided me with an escape from work and from worries. They never doubted me, and always supported me, even while going through professional and academic hardships themselves. Perhaps because I don’t have that many friends, I do cherish the ones I have very much. So, I want to tell you all how much your company, your smiles, and your hugs meant to me. May I be as good a friend to you in return. Thank you all, for everything.

I have, of course, to thank the gang (you know who you are), for keeping me company every week. For all those times we spent just talking and sharing, and helping each other with our problems. Suffice to say, I am happier for having met you all, for having shared those experiences with you. And I hope I also helped you in return, made your lives just a little bit easier. Here's to our future! Thank you all.

And finally, most of all, I'd like to thank Maria João. You, who through it all have been my greatest ally, my most trusted confidant, my stalwart companion, and dare I say my dear friend. You were there for my highs and my (really low) lows, academic or otherwise. You saw me at my worst. And you didn't mind, you didn't judge, you just kept smiling, showing me that warm smile of yours that said things would get better. And they did, eventually. They're not where I want them to be yet, but things did change. And I owe all of that to you. Today I say, with confidence, I am a better person for having met you, Maria João. Know that I admire you so, so much, and I hope, that one day, I'll get to be a person as wonderful as you. Now and forever, I am eternally grateful. Thank you for all the times we spent together. Thank you for everything you said and I stored in my heart. Thank you, Maria João, for not giving up on me.

Abstract

The ubiquitous nature of today's multi-core processors means concurrency is ever more important to effectively use available computing resources. By its very nature however, concurrent programming is complex and error-prone - accounting for random process interleavings and managing shared resource control is difficult and can lead to instances of incorrect behavior or deadlocks in concurrent programs. As such, ensuring the correctness of concurrent programs is of the utmost importance. Session types are a typing discipline for message-passing concurrency that is able to ensure strong compile-time correctness guarantees for concurrent programs by providing a protocols-as-types view of communication. In particular, we make use of the interpretation of intuitionistic linear logic formulas as session types, which serves as the basis for *logical session types*. These logical session types offer stronger guarantees at compile-time than simple session types. In this work we implement a (logically) session-typed functional language, along with its associated type checker, and develop a compiler for said language targeting the Go language. Our work features standard functional programming features combined with channel-based, session-typed, concurrency primitives and thread spawning. Concurrency and pure functional values are separated via a monad-like interface, in the style of Haskell. Our compilation pipeline takes a program, type checks it to ensure the absence of deadlocks and communication errors, and then translates it to valid Go code, leveraging Go's channel and lightweight thread infrastructure. The translation requires compensating the mismatch between Go's channel types and session types, which we achieve via a state machine view of session types. We showcase the expressiveness of our language via a series of examples, encoding concurrency idioms in the style of map-reduce, among others. We also perform a performance evaluation of our implementation, experimenting with different settings and testing it against a native Go implementation. We follow with a discussion of the experimental results. Finally, we end by discussing possible approaches to future work, namely in terms of compiler optimizations and increase in language expressiveness.

Keywords: Concurrency, Session Types, Logical Session Types, Compilation, Go, Functional Language, Bidirectional Type Checking

Resumo

Atualmente, a natureza ubíqua de processadores *multi-core* faz da concorrência algo cada vez mais importante para utilização eficaz dos recursos. Mas a programação concorrente é complexa e dada a erros – antecipar a execução intercalada de processos e gerir o controlo de recursos partilhados é difícil; pode levar a comportamento incorreto, ou *deadlocks* em programas concorrentes. Assim, assegurar a correção de programas concorrentes é da maior importância. Tipos de sessão são uma disciplina de tipos para concorrência baseada na troca de mensagens que é capaz de dar fortes garantias de correção em tempo de compilação para programas concorrentes, oferecendo uma visão da comunicação em termos de protocolos-como-tipos. Utilizamos a interpretação de fórmulas da lógica linear intuicionista como tipos de sessão, que serve de base para tipos de sessão lógicos que oferecem garantias mais fortes do que tipos de sessão simples. Neste trabalho implementamos uma linguagem funcional, com tipos de sessão (lógicos), juntamente com o seu *type checker*, e desenvolvemos um compilador para a dita linguagem que tem por alvo a linguagem Go. A linguagem apresenta as funcionalidades standard da programação funcional, combinadas com *thread spawning* e primitivas de concorrência baseadas em canais e tipificadas com tipos de sessão. A concorrência e valores funcionais são separados por uma interface tipo *monad*, ao estilo de Haskell. O processo de compilação recebe um programa, verifica o seu tipo para assegurar a ausência de *deadlocks* e erros de comunicação, e tradu-lo para código Go, utilizando a infraestrutura de canais e *lightweight threads* de Go. A tradução implica uma compensação da diferença entre os tipos de canais em Go e os tipos de sessão, que alcançamos através duma visão de tipos de sessão como máquinas de estados. Demonstramos a expressividade da linguagem através duma série de exemplos, implementando idiomas de concorrência como *map-reduce*, entre outros. Realizamos uma avaliação de desempenho da implementação, experimentado definições diferentes e testando-a contra uma implementação nativa em Go, discutindo depois os resultados experimentais. Por fim, propomos várias abordagens para trabalho futuro, em termos de optimização do compilador e aumento da expressividade da linguagem.

Palavras-chave: Concorrência, Tipos de Sessão, Tipos de Sessão Lógicos, Compilação, Go, Linguagem Funcional, Verificação de Tipos Bidirecional

Contents

List of Figures

List of Tables

1 Introduction

Concurrent programming is an increasingly important programming paradigm. In this world of multi-core architecture and interaction between different resources, concurrency is fundamental in making more efficient use of resources and modeling certain problems. Concurrent programming traditionally consists of the coexistence of execution threads, sharing access to memory and other resources. The simultaneous access to shared resources introduces new avenues of possible errors, from incorrect program behavior, to the presence of race conditions, to the existence of deadlocks. The prevention errors in concurrent programming is of course synonymous with lock-type mechanisms to keep threads from interfering with each other's behavior, and maintain synchronization between them. However, the use of such low-level concurrency primitives such as locks tends to lead to increased complexity in programs, keeping the door open for the introduction of errors in those programs.

Channel-based concurrency is a form of concurrent programming that aims to alleviate the intricacies of shared memory concurrency by having concurrent threads or processes exchange data via communication rather than interference over shared data structures. Typically, this form of concurrency does not require the use of low-level concurrency control mechanisms such as locks and provides easier to use, higher-level concurrency primitives. Still, there is still the possibility of errors when working with channel-based concurrency: the wrong type of message may be sent along a channel, or the communication may resolve in a deadlock. Modern programming languages such as Go or Rust provide support for channel-based concurrency, however they offer little support for compile-time verification of communication safety (absence of communication mismatches) and deadlock-freedom.

In these languages, channels are statically typed to carry payloads of a given *fixed* type, and a well-typed program is only ensured to never attempt to communicate over a channel using values of the wrong type. Since most programs need to exchange values of many different types, concurrent programs must make *coordinated* use of many different channels. This problem is aggravated by the fact that these channels are often unidirectional, further adding to the coordination needs and giving rise to programming errors that can result in deadlocks.

This is in sharp contrast with the state of the art typing disciplines for channel-based concurrency, which often ensure at compile time both communication safety and deadlock-freedom. Session types [14] are one such typing discipline, where channels are typed according to simple *communication protocols*. For instance, the session type $int \wedge bool \supset \mathbf{1}$ (using the syntax of Section 3.2) specifies a channel along which we send exactly one value of type *int*, afterwards input a *bool*, and then stop using. In a language with session types, type checking enforces that the channel is used according to this specification, effectively eliminating communication mismatch errors. In session type systems based on linear logic [42] (such as those this work is based on), typing further ensures that no deadlocks occur.

Session types are generally absent from general purpose languages since they require so-called *linear* typing, in order to track the stateful nature of a channel's type, where the valid payload type of a channel at a given moment depends on the previous actions taken on the channel. This results in implementations realized as DSLs or API-generation libraries (e.g. [16, 44]) which provide diminished compile-time guarantees (linear use of resources, communication safety, deadlock-freedom), offering them instead at runtime; or in special purpose languages, designed specifically with session types in mind (e.g. [1]), but lacking many of the modern features and ecosystems of fully-fledged general purpose languages. Performance is usually a further issue, since these languages are often *interpreted* rather than *compiled*.

In this work we introduce a session-typed functional language, following [42], which provides compile-time guarantees of absence of communication errors and deadlocks. Unlike other natively session typed languages, we develop a *compiler* from our language to valid Go code, fully taking advantage of Go's channels and lightweight threads. Thus, we can provide the compile-time correctness assurances of session typing, while alleviating many of the limitations of having a special purpose language.

With this work, we offer the following contributions:

- A functional, natively session-typed language
- A complete bidirectional type checker that enforces a session type system, thus guaranteeing programs are free of communication errors and deadlocks
- A simple, early version of a language interpreter
- A full-fledged compiler that compiles code written in our language to valid, and equivalent, Go language programs
  - Compilation of language session types to equivalent Go type states
  - Compilation of the language's `fwd` construct - a process level primitive that redirects communication between two communication channels

Document structure. We start by describing the relevant scientific background in Chapter 2, and discussing related work in Section 2.2. We then define the syntax of our proposed language and explain its semantics in Section 3.1, and present the language's type system in Section 3.2. The type checking process of our language is explained in Section 4.1, and the compilation process is detailed in Section 4.2. We show some example programs in Section 3.3. In Chapter 5 we evaluate the performance of the proposed language, and discuss the results. Finally, in Chapter 6, we offer some final thoughts on our work and discuss possible avenues of future work.

2 Background and Related Work

Concurrent programming consists of conceiving a program where several processes (i.e. running programs) execute in parallel, competing for access to resources and cooperating towards a common goal. Processes communicate and synchronize among themselves by either reading and writing to a shared memory space, or by exchanging messages with each other; these approaches to process communication are know as *shared memory* and *message passing*, respectively.

The concurrent programming paradigm is increasingly important for several reasons:

- Concurrent computing maximizes a device's useful processing time, allowing for the concurrent execution of some processes while others wait for slow input/output operations to finish, which minimizes wait times for users in the case of time-sensitive applications;

- Modern applications have to communicate with several heterogeneous computational resources (e.g., web servers or authentication servers) simultaneously, which must necessarily be managed in a concurrent way;

- The only way to take full advantage of today's ever more ubiquitous multi-core architectures is to write concurrent programs; the execution of a program's various processes is distributed among the various available processing cores.

Concurrent programming is generally more complex when compared to more traditional sequential programming because it's harder to reason about and assert program correctness in the face of concurrent processes. The myriad possibilities of process interleavings cause an explosion in the number of possible execution states. Besides, shared resources require fine-grained usage protocols which must be respected to ensure correct execution. Possible protocol-violating interactions between concurrent processes open new possible execution paths to incorrect program states, which range from simply incorrect or unexpected program behavior, to execution *deadlocks*, where all processes are simultaneously waiting for another process to release some shared resource, to *race conditions*, where execution depends on which process reaches a critical execution state first, leading to unpredictable behavior. Besides, concurrent programs are often hard to debug, due to the random nature of thread interleaving making bugs hard to reproduce.

The notable increase in the presence of the concurrent paradigm in modern programming makes it ever more important to assure correct, error-free, program behavior. As such, a major goal when designing concurrent programs is making sure that the interaction between concurrent processes is synchronized in a way that not only guarantees *safety* (nothing bad happens), but also ensures *liveness* (something good eventually happens). Concurrent programs present two types of **synchronization**: *mutual exclusion* - ensuring that no two processes execute a critical section of the program at the same time - and *conditional synchronization* - ensuring that a process delays its execution until a given condition is true.

Various mechanisms have been developed to specify concurrent execution, communication and synchronization [2]. In the case of shared memory approaches there are mechanisms such as locks, semaphores and monitors, that can be used by a program to enforce exclusion over certain variables/resources between different processes. In the case of message passing approaches, the various processes share channels which are abstractions of a communication link of some sort. It is common to define operations - called message passing primitives - over these channels. There are usually two main primitives: *send* and *receive* (besides other selective communication primitives like *selective wait* or *selective read*, for example). These primitives enforce the goal of synchronization between processes since a message can not be received before it is sent. In this way, the message passing technique seeks to presents a higher-level approach than the shared memory technique, looking to remove the complexity associated with using low-level concurrency primitives to coordinate concurrent access to shared memory.

2.1 Session Types

In general, languages offer little support for static verification of correct concurrency usage protocols. Though some frameworks exist or have been proposed, most languages still do not offer type-level guarantees of the absence of errors in concurrent programs. *Session type* [14] systems were some of the first proposed to address this problem. They attempt to establish a structuring method to communication-based concurrent programming; namely, they are meant to offer a basic means of describing series of reciprocal interactions between processes at a high-level of abstraction. In fact, conventional communication patterns such as remote procedural-call and method invocation can be expressed as sessions of reciprocal interactions [14]. The central idea in session type systems is that of a *session*. A session is a series of (potentially recursive) reciprocal dyadic (binary) interactions, possibly with branching, that serves as a unit of abstraction for describing process interaction. Communications belonging to that session are done via a channel specific to that session - a private channel is generated when initiating each session. The earliest session type system [14] is shown to be expressive, since it can be used to represent several communication patterns, ranging from the simple - call-return and method invocation - to the complex - continuous interactions, unbounded interactions, and delegation of processing tasks to other processes via channel-passing. As an example, we show below the syntax for typing processes proposed in [14], which makes for an adequate representation of the typical syntax of session type systems.

P, Q ::=	`request` $a(k)$ `in` P	session request
	`accept` $a(k)$ `in` P	session acceptance
	$k![\tilde{e}]; P$	data sending
	$k?(\tilde{e}); P$	data reception
	$k \triangleleft l; P$	label selection
	$k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\}$	label branching
	`throw` $k[k']; P$	channel sending
	`catch` $k(k'); P$	channel reception
	`if` e `then` P `else` Q	conditional branching
	$P \mid Q$	parallel composition
	`inact`	inaction
	$(\nu u)P$	name/channel hiding
	`def` D `in` P	recursion
	$X[\tilde{e}\tilde{k}]$	process variables

Starting Sessions The `request` $a(k)$ `in` P construct requests, through name a, the start of a session and the generation of a new channel k, along which said session is conducted. Process P then uses k in some manner. Dually, `accept` $a(k)$ `in` P accepts the request for the start of a session via name a, and generates a new channel k to be used in P.
Data Communication As for the $k![\tilde{e}]; P$ construct, it represents the sending of expressions $\tilde{e}$ over channel k, before continuing as process P. In a dual manner, $k?(\tilde{x}); P$ signifies the reception of some values through channel k, which are then bound to $\tilde{x}$ in process P.
Branching The construct $k \triangleleft l; P$ is used for label selection in branching behavior; a label l is sent through channel k before continuing as process P. Dually, $k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\}$ codifies the behavior of receiving a label l_i, before continuing as matching process P_i.
Channel Communication As for the case of data communication, the `throw` $k[k']; P$ construct is used to send a channel k' through channel k, before continuing as process P. Again, dually, `catch` $k(k'); P$ represents the reception a channel from k, which is bound to k' in continuing process P.

As for the other, more typical constructs, we note: conditional branching `if` e `then` P `else` Q, where a process' behavior (P or Q) is dependent on the truth value of expression e; parallel composition of processes $P \mid Q$; `inact`, which represents the lack of action; hiding of names $(\nu u)P$, which limits name u to the local scope of P; $X[\tilde{e}\tilde{x}]$, which represents simple process variables; and recursion definition, `def` D `in` P.

We show a simple example of the evolution of a session between two processes typed according to the described type system:

$$\begin{array}{rcl}
\texttt{accept}\ a(k)\ \texttt{in}\ k![1];k?(y)\ \texttt{in}\ P & | & \texttt{request}\ a(k)\ \texttt{in}\ k?(x)\ \texttt{in}\ k![x+1];Q \\
\rightarrow (\nu k)(k![1];k?(y)\ \texttt{in}\ P & | & k?(x)\ \texttt{in}\ k![x+1];Q) \\
\rightarrow (\nu k)(k?(y)\ \texttt{in}\ P & | & k![x+1];Q) \\
\rightarrow (\nu k)(P[2/y] & | & Q)
\end{array}$$

The process on the right first requests a session on channel k; it then receives an integer through k, bound to x, before sending back $x + 1$ through k. The process on the left behaves dually, first accepting a session on channel k, then sending 1 through channel k, before receiving value 2 through k which is bound to y in continuation P.

We also present an example of recursion, a process which provides a continuous stream of growing integers (the process definitions are split between two lines, due to matters of space):

$$\begin{array}{lcl}
\texttt{accept}\ a(k)\ \texttt{in}\ \texttt{def}\ P(x,k) = & | & \texttt{request}\ a(k)\ \texttt{in}\ k \triangleleft next; k?(y)\ \texttt{in} \ldots \texttt{in} \\
k \triangleright \{next : k![x]; P(x+1,k) \mid end : \texttt{inact}\}\ \texttt{in}\ P(0,k) & & k \triangleleft end;\ \texttt{inact}
\end{array}$$

The left process is the process which provides the integer stream. It starts by accepting a session on channel k over name a. It then provides a choice between two alternative behaviors: either it stops the session if it receives the `end` label, or it sends value x over channel k before offering the choice again, if the `next` label is received. The right process can be thought of as a client process. It first requests a session on channel k over name a. It then sends the label `next`, before receiving a value over channel k; it repeats this behavior an arbitrary number of times before finally sending label `end` over channel k, and stopping the session.

As for the type syntax of the language, we show it below:

$$\begin{array}{lllllllllll}
S & ::= & \texttt{nat} & | & \texttt{bool} & | & <\alpha,\overline{\alpha}> & | & s & | & \mu s.S \\
\\
\alpha & ::= & \downarrow[\tilde{S}];\alpha & | & \downarrow[\alpha];\beta & | & \&\{l_1:\alpha_1,\ldots,l_n:\alpha_n\} & | & 1 & | & \perp \\
 & | & \uparrow[\tilde{S}];\alpha & | & \uparrow[\alpha];\beta & | & \oplus\{l_1:\alpha_1,\ldots,l_n:\alpha_n\} & | & t & | & \mu t.\alpha
\end{array}$$

In which S represents *sorts* (the basic, or primitive types of data, so to speak), and α denotes the type of interaction that takes place over a channel. The only sort of note is $<\alpha,\overline{\alpha}>$, which represents two structures of interaction, complementary in nature, associated with a name (one which denotes the behavior starting with `accept`, the other starting with `request`). The construct $\downarrow[\tilde{S}];\alpha$ means inputting a value of sort $\tilde{S}$, and continuing by doing the actions in α; dually, $\uparrow[\tilde{S}];\alpha$ means outputting a value of sort $\tilde{S}$, and continuing with behavior in α. Similarly, $\downarrow[\alpha];\beta$ and $\uparrow[\alpha];\beta$ represent similar behavior, but inputting/outputting channels first. $\&\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}$ denotes branching behavior: wait for label l_i, and continue as α_i. $\oplus\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}$ represents the sending of label l_i, before continuing as α_i. The inaction is represented by 1. $\mu t\alpha$ represents recursive behavior (execute actions in α, until $\mathtt{t}$ is found, and recur to α). As for $\perp$, it indicates that no further communication is possible at a given name.

With the syntax and semantics of sorts and types in mind, we type the processes in the sessions given as examples above. For the first example, $\mathtt{accept}\ a(k)\ \mathtt{in}\ k![1]; k?(y)\ \mathtt{in}\ P$ is of type $\uparrow [\mathtt{nat}]; \downarrow [\mathtt{nat}]; T$, where T is the type of P; dually, $\mathtt{request}\ a(k)\ \mathtt{in}\ k?(x)\ \mathtt{in}\ k![x+1]; Q$ is of type $\downarrow [\mathtt{nat}]; \uparrow [\mathtt{nat}]; U$, where U is the type of Q. As for the recursive example, $\mathtt{accept}\ a(k)\ \mathtt{in}\ \mathtt{def}\ P(x,k) = k \triangleright \{next : k![x]; P(x+1,k) \mid end : \mathtt{inact}\}\ \mathtt{in}\ P$ is of type $\mu t.\&\{next : \uparrow [\mathtt{nat}]; t \mid end : 1\}$; in a dual manner, $\mathtt{request}\ a(k)\ \mathtt{in}\ k \triangleleft next; k?(y)\ \mathtt{in} \ldots \mathtt{in}\ k \triangleleft end;\ \mathtt{inact}$ is of type $\oplus\{next : \downarrow [\mathtt{nat}]; \ldots; \oplus\{end : 1\}\}$.

Session type systems guarantee safety of communication. And it should be noted that, while session type systems assure deadlock-freedom, they do so only in the case that two processes maintain just one session at a time; if two processes maintain two or more simultaneous sessions, the crossed session message streams give way to the possibility of deadlocks. Additionally, though (binary) session types allow for the accurate modeling of reciprocal actions, they present what may be seen as a limitation: no more than two participants can participate in a statically certified deadlock-free session. This means that there are communication patterns involving multiple participants that may not be accurately modeled by binary session types in a certifiably deadlock-free way.

2.1.1 Multiparty Session Types

To address the limitations of binary session types, *multiparty* session types [15] were proposed. Multiparty type systems introduce the notion of *global* type, which specifies participant interactions from a global or choreographic perspective. *Local* types are mechanically extracted from global types (one for each participant) and are used to type processes. If a *global* type satisfies some well-formedness properties, a multiparty session is deadlock-free. As such, multiparty session type systems address a limitation of dyadic session type systems, guaranteeing deadlock-freedom in communication between an ensemble of session participants. To better illustrate the idea of multiparty session types, we show an example of a multiparty session (as seen in Figure 2.1), and its respective global type:

Buyer1 starts by sending a book title to Seller. Seller then sends a quote to Buyer1 and Buyer2. Buyer1 notifies Buyer2 of how much it can pay. Buyer2 then informs Seller if it accepts the quote (and if so they exchange relevant information).

The syntax of global types proposed in [15] is as follows:

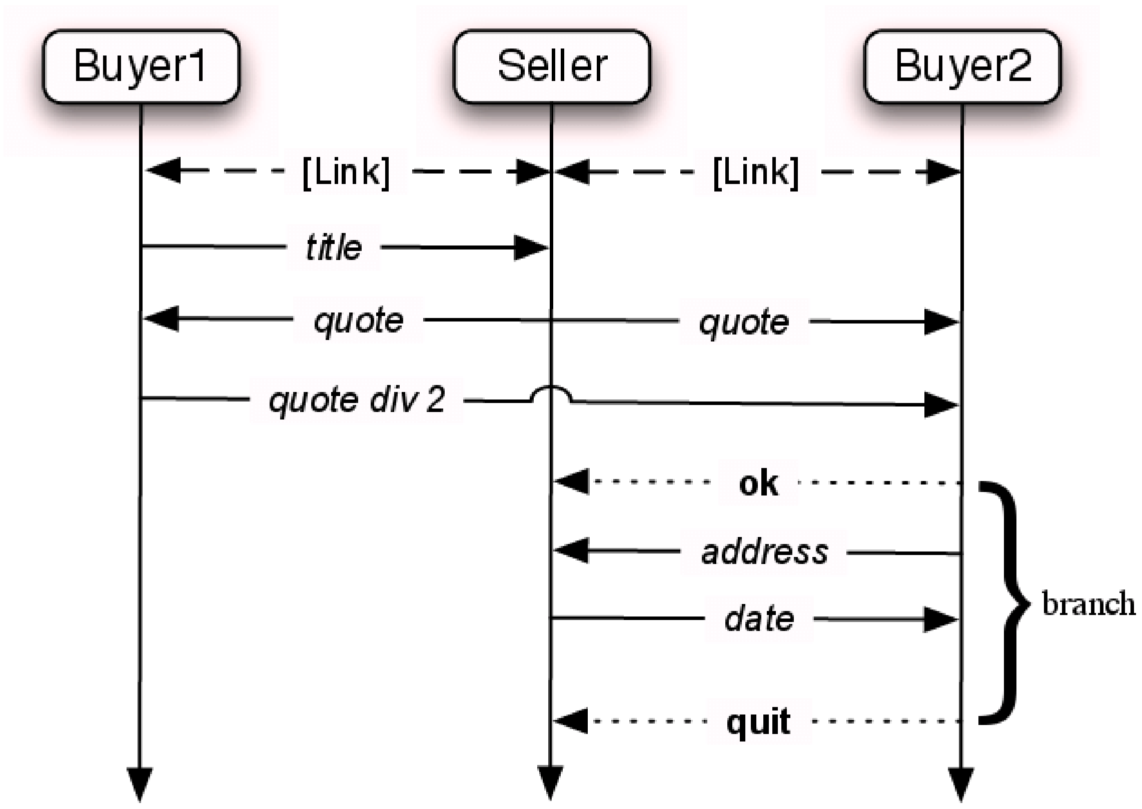

Figure 2.1: Two Buyer Protocol

$$
\begin{array}{rcll}
G & ::= & p \rightarrow p' : k < U > .G' & \text{values} \\
 & | & p \rightarrow p' : k\{l_j : G_j\}_{j \in J} & \text{branching} \\
 & | & G, G' & \text{parallel} \\
 & | & \mu t.G & \text{recursive} \\
 & | & \text{end} & \text{end} \\
 & | & t &
\end{array}
$$

Type $p \rightarrow p' : k < U > .G'$ means that process p sends value of type U to process p' along channel k and the interactions described in G' take place. As for $p \rightarrow p' : k\{l_j : G_j\}_{j \in J}$, it says that process p sends a label through channel k to process p'; if the label l_j is sent, the interactions in G_j take place. G, G' states that both the interactions in G and G' take place. $\mu t.G$ represents standard recursion, and end represents the end of a multiparty session.

Thus, the global type of the Two Buyer Protocol is the following:

$$
\begin{array}{lll}
1 & B1 \rightarrow S : & s < \text{string} > . \\
2 & S \rightarrow B1 : & b_1 < \text{int} > . \\
3 & S \rightarrow B2 : & b_2 < \text{int} > . \\
4 & B1 \rightarrow B2 : & b'_2 < \text{int} > . \\
5 & B2 \rightarrow S : & s\{\text{ok} : B2 \rightarrow S : s < \text{string} > .S \rightarrow B2 : b_2 < \text{date} > .\text{end} \ | \ \text{quit} : \text{end}\}
\end{array}
$$

There are, however, disadvantages to a multiparty session type system: assuring that a *global* type is well-formed can rule out many morally correct protocols (protocols involving recursion or message-dependencies) [38]; it's not compositional - the number of session participants is fixed and they must all synchronize globally upon session initiation; and the framework is overall complex, requiring an infrastructure of global types, local types and well-formedness checks that go beyond simple type checking.

2.1.2 Logical Session Types

Linear logic has been widely studied in regards to communicating systems, given its ability to deal with resources, effects, and non-interference [3]. Indeed, several type systems for the π-calculus, such as linear types [36], types for deadlock-freedom [19] and its refined variations [20, 21], and session types [14, 15], employ some form of linearity; however, rarely do these systems make use of linearity in a direct way and exploit the type-theoretic significance of linear logical operators, opting instead to merely exploit fine-grained type context management, or assignment of multiplicities to channels. As such, a new type system for the π-calculus was introduced [3, 32], that corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic. This *logical session type system* is based on an interpretation of intuitionistic linear logic formulas as session types, giving way to a session-typed π-calculus system in which the type structure consists of the connectives of intuitionistic linear logic, which retain their standard proof-theoretic interpretation. It is this logical session type system that our language will implement.

This type system distinguishes between two kinds of type environments: a *linear* type environment Δ and an *unrestricted* type environment Γ. A judgment of the system is of the form $\Gamma; \Delta \vdash P :: z : C$, with the domains of Γ and Δ being pairwise disjoint. Such a judgment asserts that process P provides a *safe usage* of channel z, according to the behavior specified by type C, under the assumptions of $\Gamma; \Delta$ - safe usage meaning freedom from deadlocks and communication errors.

This interpretation establishes a close correspondence between session types for the π-calculus and intuitionistic linear logic, since typing rules correspond to linear sequent calculus proof rules, and process reduction may be simulated by proof conversions and reductions, and vice versa. Sequents have the form $\Gamma; \Delta \vdash D : C$, where Γ is the unrestricted context, Δ the linear context, C a logical formula (counterpart to type) and D the proof term that represents the derivation of $\Gamma; \Delta \vdash C$. Given the parallel structure of the two systems (type system, and sequent calculus), if $\Gamma; \Delta \vdash D : A$ is derivable in the sequent calculus, then there is a process P and a name z such that $\Gamma; \Delta \vdash P :: z : A$ is derivable in the type system. The inverse result also holds true: if $\Gamma; \Delta \vdash P :: z : A$ is derivable in the type system, then there is a derivation D that proves $\Gamma; \Delta \vdash D : A$. As an example, we show below the `cut` reduction rule in dual intuitionistic linear logic, and `Tcut`, its counterpart in the type rules of the language proposed in [3]:

$$\frac{\Gamma;\Delta \vdash D : A \quad \Gamma;\Delta', x : A \vdash E : C}{\Gamma;\Delta,\Delta' \vdash cutD(x.E) : C}\ (\textsc{cut}) \qquad \frac{\Gamma;\Delta \vdash D : P :: x : A \quad \Gamma;\Delta', x : A \vdash Q : T}{\Gamma;\Delta,\Delta' \vdash (\nu x)(P \mid Q) :: T}\ (\textsc{Tcut})$$

The cut reduction rule in the sequent calculus matches synchronous communication [3]. Given that each pair of processes only shares a single channel, there are no cyclic dependencies, and therefore there are no deadlocks in communication protocols.

As a consequence of the property of *soundness* of linear logic, this type system ensures *session fidelity* - processes send and receive data correctly, according to the type of the session channel - and provides *deadlock-freedom* guarantees for systems that interact on an arbitrary number of sessions; a great improvement over the restricted property of progress on a single session obtained in the original session type systems. Logical session types offer several advantages over other kinds of session type systems: it's impossible to write a well-typed program under a logical session type system that becomes deadlocked; it's possible to express deep semantic properties through the method of logical relations [32], such as termination, behavioral equivalence, confluence and parametricity; moreover, logical session type systems are compositional.

2.2 Related Work

One major obstacle to the adoption and integration of session type systems into mainstream programming languages is their mandatory tracking of the linear usage of resources, whose built-in support would require massive changes in both languages and their development tools. Thus, several efforts have been made to implement session types in general-purpose-languages in different practical ways. These implementations can be broadly divided into two categories: implementations on top of existing general-purpose languages; and implementations on top of a specially created language. We overview both kinds of implementations and the trade-off across all of the presented implementations is their inability to provide strong static guarantees about program safety and program liveness.

2.2.1 Session Types in General Purpose Languages

Integrating session types in general purpose languages is challenging due to the need to enforce a linear typing discipline (i.e making sure resources are used once and only once) in a non-linear typing system. Thus, most implementations in this setting manifest as libraries which forego most of the compile-time correctness guarantees of session types (notably linear use of resources), only providing runtime correctness assurances when the library API is used correctly, but with no compile-time means of ensuring the usage is correct. This kind of implementation tends to be lightweight, since there is no need for additional user effort besides importing the library's features into a program. The pragmatic library approach is often complemented by a protocol description Domain-Specific Language (DSL), from which the API is generated, and some type of protocol validator. Through the description of protocols, these implementations are able to describe said communication protocols in a more fine-grained manner than plain library implementations. Note, however, that this increase in expressiveness is accompanied by a notable disadvantage: changes to an existing protocol imply changes to the generated API, breaking compatibility with existing code. Plus, despite the increased specificity, these implementations are still only capable of offering correctness guarantees at *runtime*. The library approach has been used in the context of Scala [37, 39, 40], Java [16, 44], F# [29], Python [28], OCaml [31] and Rust [18, 22, 24].

When the type system of the host language is powerful enough, linearity can be *statically* encoded using type-level programming features. This has given rise to (partially or completely) statically verified implementations in OCaml [17], Haskell [23, 25, 27, 30, 35], Java [43] and, more recently, Rust [5]. These implementations often suffer from usability issues due to the type-level encoding of linearity giving rise to inscrutable error messages [5, 35], as well as suffering from a lack of integration with the language's development ecosystem, which generally is not suited to interact with type-level programming features used in the implementations. To better illustrate the kind of error messages produced by this kind of implementations, take the language proposed in [5]. The following is an example of a program `hello_client` with a session that receives a channel on which a value is received, before closing:

```
let hello_client: Session<ReceiveChannel<ReceiveValue<String, End>, End>>
= receive_channel(| a | {
   send_value_to(a, "Alice".to_string(),
      wait(a, terminate())
) });
```

If we were to wait for the termination of *a* before sending a value to it, like so:

```
let hello_client: Session<ReceiveChannel<ReceiveValue<String, End>, End>>
= receive_channel(| a | {
      wait(a, terminate())
) });
```

We would get the following error: *the trait 'ContextLens<(ReceiveValue<String, End>, ()), End, Empty>' is not implemented for 'Z'*, which is not easily interpreted from a user standpoint.

2.2.2 Natively Session Typed Languages

Generally, languages natively featuring session types are developed in an academic setting (such as our own), and so they tend to be minimalist, and their programs are often *interpreted* rather than *compiled*, since the emphasis is on the implementation of the type-checker. We highlight the language of [11], which features *refined* session types - a form of dependent types that allow for the further specification of session types by restricting the set of values of a given type (stating such restrictions through logical formulae). This interpreted language is based on a classical formulation of session types [14] and so has weaker static correctness properties when compared to our own. This is also the case with the more recent FreeST language [1] which is based on a context-free extension of session types. Both languages do not enforce deadlock-freedom by typing, only absence of communication errors.

The line of work around the session-based functional languages SILL [33, 42] and Rast [8, 9, 7] is the most closely related to ours. Both languages are interpreted. In the former, the implementation focuses on a polarized view of session types which uniformly incorporates synchronous and asynchronous communication. In the latter, the emphasis is on extensions to the core session typing discipline that enable automated amortized parallel complexity analysis. In terms of compiled languages in this space, we highlight Concurrent C0 [47], a type-safe C-like language, with session-typed communication over channels. Concurrent C0 compiles to C or Go, but features a form of asynchronous communication requiring a specialized runtime that is absent from our communication model which can be faithfully modeled using Go's channel-based primitives directly.

3 Language

Our language consists of a session-typed functional language in the style of the SILL family [42, 41, 33] of languages based on the propositions-as-types interpretation of intuitionistic linear logic as a session-typed language [3]. The language features channel-based concurrency combined with standard functional programming features. In our language, concurrency primitives form a sub-language which we dub the *process* layer, and our typing discipline enforces a strict separation between the standard functional connectives (the *functional* layer) and the process layer, reminiscent of monadic programming in Haskell [45].

Terms in the process layer, often referred to as *processes*, may freely use terms from the functional layer (e.g. it is possible to communicate functional values). The functional layer, on the other hand, can only depend on process terms in a controlled way, so as to enforce the strong static correctness of the overall language. As we detail below, processes are *embedded* in the functional layer, becoming possible to define processes via functions.

3.1 Syntax

The syntax of our language is given in Figure 3.1. We range over functional terms with M, N and over process terms with P, Q. A program in our language is a sequence of declarations of the form `let` $x : T = N$, where the name x (of type T) can occur in N, in order to allow for recursive (function) definitions. We distinguish the top-level declaration of name `main`, which acts as the program's entry point. We also allow for (session) type declarations.

Functional terms form a standard functional language with the expected constructs: multi-argument λ-abstractions take the form `fun` $\overline{x} \longrightarrow M$ `end` (we often omit type annotations in λ-bound variables due to our use of bidirectional type-checking – Section 4.1); function application is noted as $M\ N$; potentially recursive let-bindings are written `let` $x = M$ `in` N; basic values (e.g. numbers, strings, booleans) and their operators (e.g. arithmetic, relational operators, etc.), are abstracted by the meta-variable V; branching is written `if` M `then` N_1 `else` N_2 `endif`. We highlight the construct $c \leftarrow \{P\} \leftarrow \overline{d}$

$$\begin{array}{lcl} M,N & ::= & V \mid \texttt{fun}\ \overline{x} \rightarrow M\ \texttt{end} \mid \texttt{let}\ x = M\ \texttt{in}\ N \mid M\ N \\ & \mid & \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2\ \texttt{endif} \mid c \leftarrow \{P\} \leftarrow \overline{d} \\ P,Q & ::= & \texttt{send}\ c\ M\ ;P \mid \texttt{send}\ c\ d\ ;P \mid x : T \leftarrow\ \texttt{recv}\ c\ ;P \mid \texttt{close}\ c \\ & \mid & \texttt{wait}\ c;P \mid \texttt{fwd}\ d\ c \mid c \leftarrow \texttt{spawn}\ M\ \overline{d};P \mid \texttt{case}\ c\ \texttt{of}\ \overline{l_j : (P_j)} \\ & \mid & c.l;P \mid \texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif} \mid \texttt{print}\ M;P \end{array}$$

Figure 3.1: Syntax of Expressions (M, N) and Processes (P, Q)

which internalizes a process term P as an *opaque* functional value (i.e., no evaluation of P takes place), where the process P *offers* some session behavior on channel c, while *using* channels $\overline{d}$ (both c and $\overline{d}$ are bound in P).

The process constructs codify the session behavior usage on the appropriate channels. For instance, `send` $c\ M\ ;P$ denotes a send action on channel c of functional term M (which will be fully evaluated before communication), with continuation behavior given by process P. Dually, $x : T \leftarrow$ `recv` $c\ ;P$ performs an input action on channel c, binding the received value (of type T) to x in the continuation P. For the sake of conciseness we overload send and receive operations, using similar syntax for communication of functional data and for (higher-order) communication of channels (note `send` $c\ d\ ;P$ denotes an output action in c of a channel d). In practice, there is a distinction in syntax between the two types of communication. The construct `close` c signals that no further communication will take place on channel c, whereas construct `wait` $c;P$ waits for the closure of session channel c before continuing as P. The process construct `fwd` $d\ c$ establishes a forwarder between channels c and d (which must be of the same type), essentially redirecting inputs and outputs on c to d and vice-versa.

As is usual in session-based communication, we allow selection and branching constructs: the construct `case` c `of` $\overline{l_j : (P_j)}$ denotes an *external* choice on channel c, and so the corresponding process will receive on c some label l_i and proceed according to the continuation process P_i (typing ensures that all possible branching options are covered); dually, the construct $c.l;P$ signals on channel c the selection of the branch labelled by l, with continuation P.

A process behavior may also branch by itself through `if` M `then` P `else` Q `endif`. The process will continue as P or Q depending on the value of M.

As a way to receive readable output from a process, we added the construct `print` $M;P$. A process prints the (evaluated) term M before proceeding as P.

Finally, we note the construct $c \leftarrow$ `spawn` $M\ \overline{d};P$, which is the process-level dual of the process embedding (functional) construct $c \leftarrow \{P\} \leftarrow \overline{d}$. While the latter embeds processes as values in the functional layer, the former allows for the *usage* of such values in other processes. Specifically, the execution of process $c \leftarrow$ `spawn` $M\ \overline{e};P$ will eventually evaluate M to an embedded process of the form $c \leftarrow \{Q\} \leftarrow \overline{d}$ (guaranteed by type safety), where channel c is bound in the continuation P and $\overline{e}$ denotes a list of (free) channels that will instantiate the channels $\overline{d}$ in Q. Process P and Q will subsequently execute in parallel, sharing the channel c for inter-process communication. For ease of writing, the expression M in $c \leftarrow \texttt{spawn}\ M\ \overline{e}; P$, which evaluates to process $c \leftarrow \{Q\} \leftarrow \overline{d}$, may be written directly as $\{Q\}$, where c and $\overline{d}$ are left implicit.

The example below showcases a program combining several features of our language. First, we define a *recursive* session type dubbed `IntStream`, which codifies the behavior of emitting an infinite stream of integers. Function `nats` takes an integer `n` and produces a process offering such an `IntStream` on channel `c`. The process sends the number `n` on `c` and afterwards *spawns* a recursive call of `nats (n+1)` on channel `d`, which is then connected to the offering channel `c` via the forwarding construct `fwd` d c. Thus, we can define recursive *processes* via a combination of recursive *functions*, spawn and forwarding:

```
stype IntStream = rec x. int ^ x;
let nats : int –> {IntStream} =
fun n –> c <– {
  send c n;
  d <– spawn (nats (n+1));
  fwd d c
} end;
```

The example above further showcases how process expressions in our language may execute. While embedded processes are values, we treat the function `main`, which must have a process value type, as the top-level function which may execute a process (similar to how `main` in Haskell is the function that actually triggers the execution of effectful computation).

The following examples illustrate the use of choice in our language:

```
stype IntCStream =
  rec x.&{next: int^x, stop: 1};
let augNats : int –> {IntCStream} =
fun n –> c <– {
  case c of
    next: send c n;
          d <– spawn (augNats (n+1));
          fwd d c
    stop: close c
} end;
```

```
c <– { d <– spawn {
          d.l1
          send d (5*3);
          close d
       };
       case d of
        l1: v <– recv d;
            wait d;
            close c
        l2: send d 10;
            send d 100;
            wait d;
            close c
}
```

The example on the left shows how the previous `nats` function can be augmented

$$
\begin{array}{lll}
D & ::= & \texttt{stype}\ N = A \\
T, S & ::= & \mathsf{Num} \mid \mathsf{Bool} \mid \cdots \mid T \rightarrow S \mid \{\overline{B} \vdash A\} \\
A, B & ::= & A \multimap B \mid A \otimes B \mid T \wedge B \mid T \supset B \mid \&\{\overline{l_i : A_i}\} \mid \oplus\{\overline{l_i : A_i}\} \mid \mu X.A \mid X \mid N \mid \mathbf{1}
\end{array}
$$

Figure 3.2: Functional Types (T, S), Session Types (A, B) and Declarations (D)

from an infinite stream of integers to one that can be potentially finite. The process that repeatedly communicates along `c` now offers a choice of two behaviors: `next` and `stop`. If `next` is received on `c`, the process will behave as `nats` and the stream will emit its next number. If `stop` is received, the process will close its channel and terminate. The example on the right shows both selection and branching by spawning a process which emits label `l1` to identify its subsequent behavior, in parallel with a process that inputs either `l1` or `l2`, with the `l1` branch matching the (dual) behavior of the spawned process.

3.2 Typing

The syntax of types for our language is given in Figure 3.2, following [42]. We distinguish between functional types, ranged over by T, S, which type functional terms, and session types A, B which type *channels* used in process terms. Our language also supports session type declarations (D) for convenience. Functional types are mostly standard, consisting of basic data types such as numeric types and booleans and function types $T \rightarrow S$. We note the type $\{\overline{B} \vdash A\}$, which types a functional term of the form $c \leftarrow \{P\} \leftarrow \overline{d}$, where process P will *offer* session type A on channel c, using channels $\overline{d}$ according to types $\overline{B}$.

Session types characterize sequences of communication actions that take place on channels: the type $A \multimap B$ denotes a channel on which its offering process expects to *receive a channel* typed with A to then behave according to the session type B; dually, type $A \otimes B$ denotes a channel on which its offering process will send a channel of type A and then behave according to type B; similarly, types $T \supset B$ (concrete syntax T => B) and $T \wedge B$ (concrete syntax T ^ B) denote input and output of functional values of type T with continuation type B, respectively; type $\&\{\overline{l_i : A_i}\}$ denotes a channel along which its offering process will receive one of the labels l_i and then offer the behavior prescribed by session type A_i; dually, session type $\oplus\{\overline{l_i : A_i}\}$ denotes a channel along which one sends one of the labels l_i to then behave according to A_i; type $\mathbf{1}$ denotes the inactive session; and type $\mu X.A$ and type variables X allow for recursive session types. Note that named types N may be used accordingly.

Our language features two typing judgments, written $\Psi \Vdash M : T$ and $\Psi; \Delta \vdash P :: c{:}A$. The former states that functional term M has type T under the typing assumptions of the form $x{:}T$ for free variables, tracked in context Ψ. The latter states that process P offers *session type A* along channel c by using the session behaviors specified in the *linear* typing context Δ and under the (functional) typing assumptions Ψ.

We now present the typing rules of our language, starting with those for the functional constructs (we list the typing rules for our language in Figure 3.3):

$$\frac{}{\Psi, x : T \Vdash x : T}\ (\textsc{var})$$

Trivially, the term x is of type T, if the free variable x exists in context Ψ with type T.

$$\frac{\Psi, \overline{x : T} \Vdash M : S}{\Psi \Vdash \texttt{fun}\ \overline{x} \rightarrow M\ \texttt{end} : \overline{T} \rightarrow S}\ (\textsc{fun})$$

The term $\texttt{fun}\ \overline{x} \rightarrow M\ \texttt{end}$ has type $\overline{T} \rightarrow S$, provided the functional term M has type S in a context containing $\overline{x : T}$ type assumptions. That is, the function body M is evaluated in a context where function arguments $\overline{x}$ are associated to their respective types $\overline{T}$.

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

A term of the form $\texttt{let}\ x = M\ \texttt{in}\ N$ is typed T provided M is of type S, and N is of type T in a context containing the type assumption $x : T$.

We follow with the typing rule for the construct that embeds processes in functional values:

$$\frac{\Psi; \overline{d : B} \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P\} \leftarrow \overline{d} : \{\overline{B} \vdash A\}}\ (\{\}\text{-I})$$

The rule above states that the *functional* term $c \leftarrow \{P\} \leftarrow \overline{d}$ has type $\{\overline{B} \vdash A\}$ provided its underlying process P is well-typed in a context where it will *use* channels $\overline{d}$ with types $\overline{B}$ and offer type A along channel c. Dually, we may use such values within other processes as codified by the following rule:

$$\frac{\Psi \Vdash M : \{\overline{B} \vdash A\} \qquad \Delta' = \overline{d : B} \qquad \Psi; \Delta, c{:}A \vdash P :: f{:}C}{\Psi; \Delta, \Delta' \vdash c \leftarrow \texttt{spawn}\ M\ \overline{d}; P :: f{:}C}\ (\{\}\text{-E})$$

The rule above captures the spawning of the (process) value M in parallel with process P by first typing M as a process value of type $\{\overline{B} \vdash A\}$. Since such a process must be provided with sessions $\overline{B}$, spawning it requires satisfying this constraint with some available channels $\overline{d}$, which will be consumed by the process. This is captured by isolating the context region Δ' which satisfies this constraint and is no longer available for use by process P. Having satisfied the constraints required to run M, the process P is then warranted in interacting with such a process via the channel c, which will be offered at type A. The below example shows the spawning of a process that uses other processes:

```
a <- {
  d <- spawn { x:int <- recv d; close d };
  e <- spawn { x:int <- recv e; close e };

  c <- spawn { send d 1; send e 2; wait d; wait e; close c };

  wait c;
  close a
}
```

The main process, which offers a session on channel a, begins by first spawning a process offering a session on channel d and then spawning a process offering a session on channel e. When another process is to be spawned, the one that offers a session over c, the main process passes the first two spawned processes as arguments in the spawning. The processes that offer sessions over d and e are no longer available to the main process. The most recently spawned process will communicate with the processes it received when it was spawned , sending to each of them an integer, and waiting for their channel closure, before terminating itself. The main process then waits for said process' termination before shutting down communication.

The typing rules for process terms codify how to *use* and *offer* sessions at a given type. The offered session is available on the distinguished right-hand side channel, which we often write as c. The used channels are tracked in the context Δ. As a kind of special case we have the *forwarding* construct:

$$\frac{}{\Psi; d{:}A \vdash \texttt{fwd}\ d\ c :: c{:}A}\ (\text{FWD})$$

Given a single ambient channel d of a given type A, we can *use* it to offer a behavior of the same type along c by essentially forwarding all messages between the two channel endpoints (e.g. messages sent along d are redirected to c and vice-versa – see Section 4.2.2 for more on how the forwarding behavior is implemented).

Value communication is typed by the following rules:

$$\frac{\Psi \Vdash M : T \qquad \Psi; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash \texttt{send}\ c\ M\ ; P :: c{:}T \wedge A}\ (\wedge\text{-R}) \qquad \frac{\Psi, x{:}T; \Delta, c{:}A \vdash P :: d{:}D}{\Psi; \Delta, c{:}T \wedge A \vdash x \leftarrow\ \texttt{recv}\ c\ ; P :: d{:}D}\ (\wedge\text{-L})$$

$$\frac{\Psi, x{:}T; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash x \leftarrow\ \texttt{recv}\ c\ ; P :: c{:}T \supset A}\ (\supset\text{-R}) \qquad \frac{\Psi \Vdash M : T \qquad \Psi; \Delta, d{:}A \vdash P :: c{:}C}{\Psi; \Delta, d{:}T \wedge A \vdash \texttt{send}\ d\ M\ ; P :: c{:}}\ (\supset\text{-L})$$

Rule ($\wedge$-R) allows us to type a process term that *offers* a session of type $T \wedge A$ by sending a term M of type T along channel c and then offering the behavior A along c. Dually, to *use* such a session we must *receive* a value of type T, bound to x in the continuation process P, which may then subsequently use the channel as type A. The remaining typing rules for process terms follow a similar pattern, the so-called *right* rules (marked with R) codify typing of processes offering sessions, and *left* rules (marked with L) codify typing of processes using ambient sessions of a given type (as is the case for rules ($\supset$ R) and ($\supset$ L), which type processes offering and using sessions of type $T \supset A$, respectively.)

The rules for typing channel communication are very similar to those of value communication:

$$\frac{\Psi; \Delta, x{:}A \vdash P :: c{:}B}{\Psi; \Delta \vdash x \leftarrow \texttt{recv}\ c\ ; P :: c{:}A \multimap B}\ (\multimap\text{-R}) \qquad \frac{\Psi; \Delta, d{:}B \vdash P :: c{:}C \qquad \Delta' = e{:}A}{\Psi; \Delta, \Delta', d{:}A \multimap B \vdash \texttt{send}\ d\ e\ ; P :: c{:}C}\ (\multimap\text{-L})$$

$$\frac{\Psi; \Delta \vdash P :: c{:}B \qquad \Delta' = e{:}A}{\Psi; \Delta, \Delta' \vdash \texttt{send}\ c\ e\ ; P :: c{:}A \otimes B}\ (\otimes\text{-R}) \qquad \frac{\Psi; \Delta, x{:}A, c{:}B \vdash P :: d{:}D}{\Psi; \Delta, c{:}A \otimes B \vdash x \leftarrow \texttt{recv}\ c\ ; P :: d{:}D}\ (\otimes\text{-L})$$

Rule ($\multimap$-R) types a process that offers a session of type $A \multimap B$, receiving along channel c a channel of type A that is bound to x in continuation P. Rule ($\multimap$-L) types its dual, a process that uses a session of type $A \multimap B$, offered along channel d, through which it sends an existing channel e of type A, which is removed from linear context Δ in the continuation P. Rules ($\otimes$-R) and ($\otimes$-L) work in the same vein, but for type $A \otimes B$.

Next we present the rules for branching and selection:

$$\frac{\Psi; \Delta \vdash P_1 :: c{:}A_1 \qquad \ldots \qquad \Psi; \Delta \vdash P_n :: c{:}A_n}{\Psi; \Delta \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j : (P_j)} :: c{:} \&\ \{\overline{l_j : A_j}\}}\ (\&\text{-R}) \qquad \frac{\Psi; \Delta, c{:}A_i \vdash P :: d{:}D}{\Psi; \Delta, c{:} \&\ \{\overline{l_j : A_j}\} \vdash c.l_i; P :: d{:}D}\ (\&\text{-L})$$

We show the rules for the choice type $\&\{\overline{l_j : A_j}\}$ above, where a process offers a session of type $\&\{\overline{l_j : A_j}\}$ by providing with an alternative l_i, for each of the choice labels, with the appropriate type A_i, on channel c. Using such a session requires committing to one of the possible choices l_i, which is sent over channel c and then warrants the use of c as a session of type A_i.

We also show the rules for the choice type $\oplus\{\overline{l_j : A_j}\}$:

$$\frac{\Psi; \Delta \vdash P :: c{:}A_i}{\Psi; \Delta \vdash c.l_i; P :: c{:} \oplus \{\overline{l_j : A_j}\}}\ (\oplus\text{-R}) \qquad \frac{\Psi; \Delta, d{:}A_1 \vdash P_1 :: c{:}C \qquad \ldots \qquad \Psi; \Delta, d{:}A_n \vdash P_n :: c{:}C}{\Psi; \Delta, d{:} \oplus \{\overline{l_j : A_j}\} \vdash \texttt{case}\ d\ \texttt{of}\ \overline{l_j : (P_j)} :: c{:}C}\ (\oplus\text{-L})$$

A process of type $\oplus\{\overline{l_j : A_j}\}$ offers a session that provides a list of labels $\overline{l_j}$, to each one associated a respective type from $\overline{A_j}$. The offering process itself will choose what label to send, unlike with sessions of type $\&\{\overline{l_j : A_j}\}$. The process using this session must define, for each possible label l_i, appropriate following behavior of type A_i, which it will follow once the offering process has sent its chosen label through d. Below we show examples where a process offers $\&\{rcv : Num \supset 1, snd : Num \wedge 1, end : 1\}$ (left) and $\oplus\{label : Bool \supset Num \wedge 1\}$ (right) along channel d:

```
c <- {
  d <- spawn {
    case d of
      rcv: n:int <- recv d; close d
      snd: send d 45; close d
      end: close d
    };

  d.snd;
  m:int <- recv d;
  print m;
  wait d;
  close a
}
```

```
c <- {
  d <- spawn {
    d.label;
    x:bool <- recv d;
    send d 21;
    close d
    };

  case d of
    label:
      send d true;
      y:int <- recv d;
      wait d;
      close c
}
```

In the left process, there is first a spawning of another process, which offers $\&\{rcv : Num \supset 1, snd : Num \wedge 1, end : 1\}$ along d. The choice that is offered is between three options, each one labeled differently: `rcv`, `snd`, and `end`, each one with its own behavior. The main, or outer, process (offers session over c), will select its choice of option by sending a label through d. It will then continue by receiving an integer and waiting for d's closure, which is the dual behavior to that of the chosen label. The right process first spawns a process offering $\oplus\{label : Bool \supset Num \wedge 1\}$ over d. Such process signals how it will behave by sending a label through d. The main process, which is receiving the label must have defined a list of possible cases, at least one of which must match that label. After the label exchange both processes continue, eventually reaching termination.

We present rules for closing a channel, and "consuming" closed channels:

$$\frac{}{\Psi; \epsilon \vdash \texttt{close}\ c :: c{:}1}\ (1R) \qquad \frac{\Psi; \Delta \vdash P :: d{:}A}{\Psi; \Delta, c{:}1 \vdash \texttt{wait}\ c; P :: d{:}A}\ (1L)$$

Above we show the rule for channel closing (1R). A process offers a session of type 1, signaling no further communication will take place on channel c.The dual to this is rule (1L): the process uses a session of type 1, removing channel c from its linear context Δ, before continuing as P, offering a session of type A along d.

Of note are the rules for typing recursive processes:

$$\frac{\Psi;\Delta \vdash P :: c{:}A\{(\mu X.A)/X\}}{\Psi;\Delta \vdash P :: c{:}\mu X.A}\ (\mu\text{-R}) \qquad \frac{\Psi;\Delta, d{:}A\{(\mu X.A)/X\} \vdash P :: c{:}C}{\Psi;\Delta, d{:}\mu X.A \vdash P :: c{:}C}\ (\mu\text{-L})$$

The type $(\mu X.A)$ is the recursive type. It represents the type of a process that acts according to session type A and then repeats its behavior. Rule (μ-R) states that if a process P that offers session type $A\{(\mu X.A)/X\}$ along channel c is well-typed, then it also well-typed if it offers $(\mu X.A)$ along c. The previous is true since $A\{(\mu X.A)/X\}$ is just a recursive unfolding of $(\mu X.A)$, and as such, both types are equivalent. The same logic applies in case of rule (μ-L), for a process using a session of type $(\mu X.A)$.

We also present the rule for the if branching construct:

$$\frac{\Psi \Vdash M : \texttt{bool} \qquad \Psi;\Delta \vdash P :: c{:}A \qquad \Psi;\Delta \vdash Q :: c{:}A}{\Psi;\Delta \vdash \texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif} :: c{:}A}\ (\textsc{if})$$

The term $\texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif}$ offers a session of type A along c if M is a boolean expression and both P and Q offer type A along c. Both branches of the if construct are of the same type, so that will be the resulting type, regardless of the actual value of M.

Finally, the rule for the print construct:

$$\frac{\Psi \Vdash M : T \qquad \Psi;\Delta \vdash P :: c{:}A}{\Psi;\Delta \vdash \texttt{print}\ M; P :: c{:}A}\ (\textsc{print})$$

The process prints a functional value and continues as process P. So, $\texttt{print}\ M; P$ offers a session of type A along c if the functional term M is of a type T, and the continuing process P also offers a session of type A along c.

Typing ensures that no well-typed program can result in a deadlock, as well as the absence of communication mismatch errors [42]. While all examples presented so far are well-typed (i.e., deadlock-free), processes P1 and P2 below

```
let P1 = c <- { send c 1 ;
                x <- recv d ;
                wait d ;
                close c } <- d
```

```
let P2 = d <- { send d 2 ;
                x <- recv c ;
                wait c;
                close d } <- c
```

are individually well-typed, but the parallel composition of the two, which deadlocks due to the wrong ordering of actions on channels c and d, is ill typed.

The two theorems offered in [42], those of type preservation and progress, assure us that at any point during the execution of a process, either the process is terminated and no further communication or computation will take place, or there is still further communication or computation to be made. As such, we can say for certain that the language we present guarantees deadlock-freedom.

$$\frac{\Psi, \overline{x : T} \Vdash M : S}{\Psi \Vdash \mathsf{fun}\ \overline{x_i} \longrightarrow M\ \mathsf{end} : \overline{T} \longrightarrow S}\ (\textsc{fun}) \qquad \frac{}{\Psi, x : T \Vdash x : T}\ (\textsc{var})$$

$$\frac{\Psi \Vdash M : S \qquad \Psi, x : S \Vdash N : T}{\Psi \Vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : T}\ (\textsc{let}) \quad \frac{\Psi \Vdash M : \mathsf{bool} \qquad \Psi \Vdash N_1 : T \qquad \Psi \Vdash N_2 : T}{\Psi \Vdash \mathtt{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2\ \mathtt{endif} : T}\ (\textsc{if})$$

$$\frac{\Psi; \overline{d : B} \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P\} \leftarrow \overline{d} : \{\overline{B} \vdash A\}}\ (\{\}\text{-I}) \qquad \frac{}{\Psi; d{:}A \vdash \mathsf{fwd}\ d\ c :: c{:}A}\ (\textsc{id})$$

$$\frac{\Psi \Vdash M : \{\overline{B} \vdash A\} \qquad \Delta' = \overline{d : B} \qquad \Psi; \Delta, c{:}A \vdash P :: f{:}C}{\Psi; \Delta, \Delta' \vdash c \leftarrow \mathsf{spawn}\ M\ \overline{d}; P :: f{:}C}\ (\{\}\text{-E})$$

$$\frac{\Psi \Vdash M : T \qquad \Psi; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash \mathsf{send}\ c\ M\ ; P :: c{:}T \wedge A}\ (\wedge\text{-R}) \qquad \frac{\Psi, x{:}T; \Delta, c{:}A \vdash P :: d{:}D}{\Psi; \Delta, c{:}T \wedge A \vdash x \leftarrow\ \mathtt{recv}\ c\ ; P :: d{:}D}\ (\wedge\text{-L})$$

$$\frac{\Psi, x{:}T; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash x \leftarrow\ \mathtt{recv}\ c\ ; P :: c{:}T \supset A}\ (\supset\text{-R}) \qquad \frac{\Psi \Vdash M : T \qquad \Psi; \Delta, d{:}A \vdash P :: c{:}C}{\Psi; \Delta, d{:}T \wedge A \vdash \mathsf{send}\ d\ M\ ; P :: c{:}C}\ (\supset\text{-L})$$

$$\frac{\Psi; \Delta \vdash P_1 :: c{:}A_1 \qquad \ldots \qquad \Psi; \Delta \vdash P_n :: c{:}A_n}{\Psi; \Delta \vdash \mathsf{case}\ c\ \mathsf{of}\ \overline{l_j : (P_j)} :: c{:}\,\&\,\{\overline{l_j : A_j}\}}\ (\&\text{-R}) \qquad \frac{\Psi; \Delta, c{:}A_i \vdash P :: d{:}D}{\Psi; \Delta, c{:}\,\&\,\{\overline{l_j : A_j}\} \vdash c.l_i; P :: d{:}D}\ (\&\text{-L})$$

$$\frac{\Psi; \Delta \vdash P :: c{:}A_i}{\Psi; \Delta \vdash c.l_i; P :: c{:} \oplus \{\overline{l_j : A_j}\}}\ (\oplus\text{-R}) \qquad \frac{\Psi; \Delta, d{:}A_1 \vdash P_1 :: c{:}C \qquad \ldots \qquad \Psi; \Delta, d{:}A_n \vdash P_n :: c{:}C}{\Psi; \Delta, d{:} \oplus \{\overline{l_j : A_j}\} \vdash \mathsf{case}\ d\ \mathsf{of}\ \overline{l_j : (P_j)} :: c{:}C}\ (\oplus\text{-L})$$

$$\frac{\Psi; \Delta \vdash P :: c{:}A\{(\mu X.A)/X\}}{\Psi; \Delta \vdash P :: c{:}\mu X.A}\ (\mu\text{-R}) \qquad \frac{\Psi; \Delta, d{:}A\{(\mu X.A)/X\} \vdash P :: c{:}C}{\Psi; \Delta, d{:}\mu X.A \vdash P :: c{:}C}\ (\mu\text{-L})$$

$$\frac{\Psi; \Delta, x{:}A \vdash P :: c{:}B}{\Psi; \Delta \vdash x \leftarrow\ \mathtt{recv}\ c\ ; P :: c{:}A \multimap B}\ (\multimap\text{-R}) \qquad \frac{\Psi; \Delta, d{:}B \vdash P :: c{:}C \qquad \Delta' = e{:}A}{\Psi; \Delta, \Delta', d{:}A \multimap B \vdash \mathsf{send}\ d\ e\ ; P :: c{:}C}\ (\multimap\text{-L})$$

$$\frac{\Psi; \Delta \vdash P :: c{:}B}{\Psi; \Delta, e{:}A \vdash \mathsf{send}\ c\ e\ ; P :: c{:}A \otimes B}\ (\otimes\text{-R}) \qquad \frac{\Psi; \Delta, x{:}A, c{:}B \vdash P :: d{:}D}{\Psi; \Delta, c{:}A \otimes B \vdash x \leftarrow\ \mathtt{recv}\ c\ ; P :: d{:}D}\ (\otimes\text{-L})$$

$$\frac{}{\Psi \vdash \mathtt{close}\ c :: c{:}1}\ (1\text{R}) \qquad \frac{\Psi; \Delta \vdash P :: d{:}A}{\Psi; \Delta, c{:}1 \vdash \mathtt{wait}\ c; P :: d{:}A}\ (1\text{L})$$

$$\frac{\Psi \Vdash M : \mathsf{bool} \qquad \Psi; \Delta \vdash P :: c{:}A \qquad \Psi; \Delta \vdash Q :: c{:}A}{\Psi; \Delta \vdash \mathtt{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathtt{endif} :: c{:}A}\ (\textsc{if}) \qquad \frac{\Psi \Vdash M : T \qquad \Psi; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash \mathsf{print}\ M; P :: c{:}A}\ (\textsc{print})$$

Figure 3.3: Typing rules

3.3 Examples

We illustrate the expressiveness of our language with a (simplified) implementation of a concurrent cryptocurrency miner and a map-reduce style generalization. With cryptocurrency, double-spending of currency is prevented by a so-called *proof of work* function, which is computationally hard to generate but easy to verify, which is used to ensure consistency across the operation ledgers in the network. Clients try to generate an answer to the proof of work, validating their version of the ledger, and collecting currency as they do. The problem, while computationally hard, naturally suggests a concurrent solution since the calculation is performed over (disjoint) ranges of values.

In our language, we can emulate this kind of concurrent pattern, shown in Figure 3.4, with a *server* which performs the calculation over a *range of integers*, from 0 to a given value `x`, spawning four worker processes among which it distributes the work. The worker processes compute a solution over the given range. Function `partial_solve` encodes the calculation over the range, returning an integer result that is then sent to the server. The server subsequently receives the results from the workers and aggregates them using a `solve` function, sending the result back to the client.

A related concurrent pattern that is straightforward to implement in our language is a generalization of the example above, but where clients can send *a function* to be executed by a server. A process which models such a server is shown in Figure 3.5, receiving some integer function `myfun`, which takes two integers and returns a single integer as a result. The server receives the function and both its arguments before executing it and returning the result. We omit a concurrent execution of the received function, but it is relatively easy to combine the two patterns.

More expressive examples of our language can be found in Appendix A.

```
let worker : unit -> {int => int => int ^ 1} =
fun u ->
  w <- {
    low:int <- recv w;
    high:int <- recv w;
    send w (partial_solve low high);
    close w
  }
end;
let master : int -> {int ^ 1} =
fun x ->
  c <- {
    w1 <- spawn (worker ());
    w2 <- spawn (worker ());
    w3 <- spawn (worker ());
    w4 <- spawn (worker ());

    send w1 0; send w1 (x / 4);
    send w2 (x / 4); send w2 (x / 2);
    send w3 (x / 2); send w3 (3 * x / 4);
    send w4 (3 * x / 4); send w4 x;

    res1:int <- recv w1; wait w1;
    res2:int <- recv w2; wait w2;
    res3:int <- recv w3; wait w3;
    res4:int <- recv w4; wait w4;

    send c (solve res1 res2 res3 res4);
    close c
  }
end;
let main : unit -> {1} =
fun u ->
    c <- {
      d <- spawn (master 1024);
      res:int <- recv d;
      print res;
      wait d;
      close c
    }
end;
```

Figure 3.4: Concurrent Cryptocurrency Miner

```
let server : unit -> {(int->int->int) => int => int => int ^ 1} =
fun u -> c <- {
                myfun:(int->int->int) <- recv c;
                fst:int <- recv c;
                snd:int <- recv c;
                send c ( myfun fst snd );
                close c
              }
end;
let main : unit -> {1} =
fun u ->
     c <- {
        d <- spawn ( server () );
        send d heavy_fun;
        send d 1024;
        send d 2048;

        res:int <- recv d;
        print res;

        wait d;
        close c
     }
end;
```

Figure 3.5: Remote Function Execution

4 Implementation

Our implementation, which is done in OCaml, is split in two stages: *type checking*, which consists of an implementation of the typing system introduced in Section 3.2, using the technique of *bidirectional* type checking [34, 10]; and *compilation* to executable Go code. The type checking process further augments the abstract syntax tree with reconstructed typing information that is omitted in the user-level syntax, to be used in the compilation stage. Compilation is split into two steps: the *preamble generation* step, where every session type in a program is converted into a set of types and associated methods in the host language; and the code generation step, where the instructions of the program are compiled into Go code.

4.1 Type Checking

We implement the type checker for our language by making use of the technique of bidirectional typing to reformulate the declarative typing judgments $\Psi \Vdash M : T$ and $\Psi; \Delta \vdash P :: c{:}A$ in terms of algorithmic *checking* and *synthesis* judgments.

For the functional layer, the approach is standard and so we omit the rules for the new judgments: we consider algorithmic judgments $\Psi \Vdash M \Rightarrow T$ and $\Psi \Vdash M \Leftarrow T$, which denote that M synthesizes type T and M checks against type T, respectively. Each rule in the declarative system corresponds to either a synthesis or a checking rule. The main practical consequence of this approach is that we need only have type annotations in top-level definitions and can omit type annotations in all other binders entirely.

For the process layer, the algorithmic formulation of the typing rules must also account for the linear treatment of the context Δ, where *all* ambient sessions must be fully used. We achieve this following the approach of [4], formulating the algorithmic version of the system in terms of *input* and *output* channel contexts: $\Psi; \Delta_I / \Delta_O \vdash P \Rightarrow c{:}A$ and $\Psi; \Delta_I / \Delta_O \vdash P \Leftarrow c{:}A$. The input context Δ_I identifies the channels that are necessarily used by P. The output context Δ_O tracks leftover channels that must still be used. At the top-level, meaning when typing P in functional term $c \leftarrow \{P\}$, Δ_O must necessarily be empty. The use of $\Leftarrow$ and $\Rightarrow$ to denote synthesis and checking is as in the functional setting.

For instance, the algorithmic formulation of the typing rule for process composition is:

$$\frac{\Psi \Vdash M \Rightarrow \{\overline{B} \vdash A\} \qquad \Delta_I' = \Delta_I \setminus \{\overline{d{:}B}\} \qquad \Psi; \Delta_I', c{:}A, /\Delta_O \vdash P \Rightarrow f{:}C}{\Psi; \Delta_I/\Delta_O \vdash c \leftarrow \mathtt{spawn}\ M\ \overline{d}; P \Rightarrow f{:}C}\ (\{\}\text{-E})$$

We can synthesize the type for $c \leftarrow$ `spawn` $M\ \overline{d}; P$ under input context Δ_I by first synthesizing the type of M, which allows us to calculate the input context for the typing of P from the input context Δ_I by removing from it the channels $\overline{d{:}B}$ and adding a new binding $c{:}A$. The output context for the typing of spawn is propagated from the output context for the typing of P.

The checking and synthesis modes are mediated by the following rules:

$$\frac{\Psi; \Delta_I/\Delta_O \vdash P \Rightarrow c{:}A \quad A \leq B}{\Psi; \Delta_I/\Delta_O \vdash P \Leftarrow c{:}B}\ (\textsc{sub}) \qquad \frac{\Psi; \Delta_I/\Delta_O \vdash P \Leftarrow c{:}A}{\Psi; \Delta_I/\Delta_O \vdash (P :: c{:}A) \Rightarrow c{:}A}\ (\textsc{annot})$$

The (SUB) rule incorporates session subtyping $A \leq B$ into the algorithmic system [12, 13]. In our language, subtyping is used to handle unfolding and folding of recursive session types. For instance, through the (SUB) rule, a process offering a session of type $\mu X. \& \{\mathsf{next} : \mathit{int} \wedge (\mu X. \& \{\mathsf{next} : \mathit{int} \wedge X, \mathsf{stop} : \mathbf{1}\}), \mathsf{stop} : \mathbf{1}\}$ can be treated as one offering a session of the folded type $\mu X. \& \{\mathsf{next} : \mathit{int} \wedge X, \mathsf{stop} : \mathbf{1}\}$, without the need for explicit fold or unfold annotations in the syntax. The (ANNOT) rule simply states that if a process P can be checked against a given type A, then the same type A can be synthesized from P.

Bidirectional type checking is implemented at the functional and process levels. For the functional layer there are two functions, one for checking functional values against a given type, `check`, and one for synthesizing a type from a functional value, `synth`, with the respective signatures:

```
rec check lin_ctxt env e t used_vars : (string * stype) list -> (string * ty) list ->
exp -> ty -> string list -> exp * ty * string list
```

```
rec synth lin_ctxt env e used_vars : (string * stype) list -> (string * ty) list ->
exp -> string list -> exp * ty * string list
```

Besides the OCaml native types, `string`, and `list`, there are various types which represent specific elements in our language: `ty`, the type of a functional value; `exp`, a functional value, or expression; `stype`, the type of a session offered by a process; and `proc` (see below), a process in our language. Note that several `proc` syntactical constructions (like `send` and `recv`, for instance), can be typed in one of both ways, according to typing rules in Section 3.2.

Both functions take some common arguments: `lin_ctxt`, a linear context (Δ), which is used to keep track of available channels if checking/synthesizing the spawning of a process; `env`, a functional environment (Ψ); e, the functional value to be typed; and `used_vars`, a list of used identifiers (variables), which do not impact the process of synthesizing/checking, but are required for a future compilation step (see Section 4.2).

The checking function also takes t, the type to check against. The return value is the same for both functions, a tuple containing the evaluated functional value (which is annotated with additional type information), the checked/synthesized type, and the identifiers used in the typing procedure.

At the process level only `synth_proc`, the synthesizing function exists, the checking is done through a combination of the synthesizing function and a subtyping function [12] for session type comparison. The subtyping function is required, in place of a simple equality checking function, due to the existence of recursive session types in our language. To clarify with an example, $\mu X.\ \&\ \{\mathsf{next} : int \wedge (\mu X.\ \&\ \{\mathsf{next} : int \wedge X, \mathsf{stop} : \mathbf{1}\}), \mathsf{stop} : \mathbf{1}\}$ is a subtype of session type $\mu X.\ \&\ \{\mathsf{next} : int \wedge X, \mathsf{stop} : \mathbf{1}\}$. The subtyping function works recursively, expanding recursive types during a type comparison, and keeping already compared type pairs in memory. Once the function tries to compare a pair of types that has already been compared, the function returns; one of the types is subtype of the other. For sake of completeness, before showing the process synthesizing function, we detail the procedure of correctly typing a process expression $(c \leftarrow \{P\} \leftarrow \overline{d})$ by checking it against a valid type:

```
let rec check lin_ctxt env e t used_vars =
match e with
| _ –> let (e′, t′, vars) = synth lin_ctxt env e used_vars in
if ty_eq t′ t then (e′, t′, vars) else error (UnexpectedType (t′, t))
```

First, there is a call to the `check` function; the type is synthesized from expression e. Next, t and t' are checked for equality by calling function `ty_eq`. Depending on the result of the call to `ty_eq`, either an error is thrown, or a tuple containing functional value e', type t', and used variables `vars` is returned.

```
ty_eq t1 t2 =
match t1, t2 with
| TProc (p1, ctxt1), TProc (p2, ctxt2) –>
  match_lin_ctxt ctxt1 ctxt2 && subtyping [] p1 p2
| _ –> t1 = t2
```

During the call to `ty_eq`, both types $t1$ and $t2$ are matched with `TProc`, since they are both the types of a process expression. Note that $p1$ and $p2$ represent the session types of the processes, and $ctxt1$ and $ctxt2$ represent the linear contexts that are used by the processes to offer their sessions. In order for $t1$ to be equal to $t2$, $p1$ must be subtype of $p2$ (or vice-versa, the subtyping function works both ways simultaneously), and both linear contexts lin_ctxt1, and lin_ctxt2 must be the same, i.e. they must have the same length, and for every channel-type pair (c, st) in $ctxt1$, there must also exist, in $ctxt2$, a pair with channel c, and a type st', such that st is a subtype of st' (or vice versa). These requirements are assured by functions `subtyping` and `match_lin_ctxt`, respectively.

```
rec synth_proc lin_ctxt env process c used_vars : (string * stype) list –> (string * ty) list –>
```

```
proc −> string −> string list −> (string * stype) list * proc * stype * string list
```

As for the process synthesis function, it takes as arguments: `lin_ctxt`, the linear context under which type synthesis must take place; `env`, the functional environment needed to type functional expressions (spawning processes or sending a functional value); `process`, the process to type; *c*, the name of the channel that the process is offering its session on; and `used_vars`, a collection of used identifiers for compilation purposes. As for the return value of the function, its a tuple with the output channel context, the typed process, the actual session type, and a list of used identifiers.

We present some examples from the process synthesizing function, detailing the segments responsible for typing certain processes:

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

Above is the part responsible for typing a process that sends a value. The function matches the process to be typed with a process that sends a value, at line 2. First there is a check of whether the process is sending a value over its own channel, or another channel’s process, at line 4.

If it is over its own channel, then the continuation p has its type synthesized, returning a tuple, with the output context, p', which is the same as p, but it (as well as it continuation and so on) may be tagged with additional typing information, session type st, and used variables (line 5). Next, at line 6, the functional value to be sent, the synthesis function is called for e, which returns e', again possibly enriched with typing information, its type t, and more used variables. Finally, at line 7, the function returns the output context of the synthesis, a version of $process$ supplemented with type t as well as e' and p', the session type that was synthesized $t \wedge st$, and a concatenation of all used variables.

If the process is sending a value over a channel other than its own, then, as is the case at line 9, first, the session type matching that other channel is pulled from the linear context (there is an error if no such channel-type association exists in the linear context). The session type is unfolded once (line 10), if it is a recursive type, to access the inner non-recursive session type (`unfold` function does nothing if the session type is not recursive). At line 11, the unfolded session type is matched with the type $t \supset st$ (otherwise there is a typing error), meaning there is an available channel to receive a functional value of type t. Next, the functional value e is checked against t, returning e', the augmented version of e, the type t, and a list of variables used by the functional value, at line 12. The linear context is updated at line 13; channel c' is no longer associated to session type $t \supset st$, and is linked to type st. The next step is the type synthesis of continuation p, under the new linear context, at line 14. The return value is a tuple containing the output context, type enriched p', session type st', and a list of variables used by p. At last the function returns the output linear context; a version of $process$ supplemented with e', type t, and p'; the type of $process$, and a concatenation of all used variables for compilation purposes (line 15).

```
1  rec synth_proc lin_ctxt env process c used_vars =
2  begin match process with
3  | Close (c') ->
4  if c' = c then
5    (lin_ctxt, process, STEnd, used_vars)
6  else
7    error (CannotCloseUnownedChannel c')
8  end
```

The synthesis function above matches the process to type to a process that closes its channel and shuts down communication (line 3). If the process is attempting to close its own channel (line 4), the function will return the linear context (unused), the typed process itself, the session type 1, and the list of used variables (line 5). If the process is attempting to close a channel other than its own then there is a typing error.

```
 1  rec synth_proc lin_ctxt env process c used_vars =
 2  begin match process with
 3  | Wait (c', p) ->
 4  if c' = c then
 5    error (CannotWaitOwnChannel c')
 6  else
 7    begin match List.assoc_opt c' lin_ctxt with
 8    | Some folded -> begin match unfold folded with
 9      | STEnd -> let new_lin_ctxt = List.remove_assoc c' lin_ctxt in
10        let (out_ctxt, p', st, vars) = synth_proc new_lin_ctxt env p c used_vars in
11          (out_ctxt, Wait (c', p'), st, vars)
12      | st -> error (CannotWaitChannelOfType (c', st))
13      end
14    | None -> error (NoSuchChannelInContext c')
15    end
16  end
```

The above function matches, at line 3, the process to one waiting on a given channel c', before continuing as p. If the process tries to wait on its own channel (line 4), there is a typing error, since that is impossible. Otherwise, the type matching channel c' is taken from the linear context and unfolded (lines 7-8), due to the possibility of being a recursive session type. If there is no matching type in context or the matching type is not 1, then an error is thrown. Given that the process is waiting on a channel matching type 1, the linear context is altered to remove the c'-1 type association, and there is the synthesis of the session type for continuation p (lines 9-10). The result, at line 11, is the output context, a type labeled p', session type st, and a list of variables used in p. The function returns the output context, a process that waits on channel c' and continues as p', type st, and the list of used variables for the purpose of compilation.

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

The synthesis function above matches, at line 3, the *process* to a process that spawns another process on channel $c'$from functional value e, with continuation p, and taking as context the channels in *args*. If the process attempts to spawn on an existing channel (be it because there is already such channel in the linear context (line 3-4), or the channel is the process' own (line 7-8)), an error is thrown. The first step, at line 10, is splitting the linear context into disjoint contexts; one context, `spawn_exp_context` to pass to the synthesis function that types the spawned process from the functional value, and another context, `next_lin_ctxt` that will be modified and passed to the application of the synthesis function to continuation p. After the context split, there is the aforementioned synthesis of e, with the specific `spawn_exp_context`, which returns the labeled e', type t, and a list of variables used in e (line 11). If type t does no match a process type (trying to spawn something that is not a process), an error is thrown, at line 17. Afterwards, at line 14, the association of c' with type st is added to `next_lin_ctxt`, to form a new linear context, `new_lin_ctxt`. There is the synthesis of the type of continuation p, at line 15, which returns the output context, type augmented p', st', and a list of variables used in p. Finally, at line 16, the function returns: the output context, a process, with continuation p', that spawns, from e', on channel c', using channels in *args*, another process of type st; session type st'; a list of all used variables.

```
 1  rec synth_proc lin_ctxt env process c used_vars =
 2  begin match process with
 3  | Choice (c′, l) –> if c′ = c then
 4  let (ctxt_opt, typed_pairs, proc_pairs, vars) =
 5  synth_external_choice_proc_list lin_ctxt env c l used_vars in
 6    begin match ctxt_opt with
 7    | None –> error EmptyChoice
 8    | Some ctxt –> (ctxt, Choice (c′, proc_pairs), STExtChoice typed_pairs, vars)
 9    end
10  else begin match List.assoc_opt c′ lin_ctxt with
11  | Some folded –> begin match unfold folded with
12    | STIntChoice choice_pairs –>
13      let (ctxt_opt, st_opt, new_procs, vars) =
14      synth_internal_choice_proc_list lin_ctxt env c c′ l choice_pairs used_vars in
15        begin match ctxt_opt with
16        | None –> error EmptyChoice
17        | Some ctxt –> begin match st_opt with
18          | None –> error EmptyChoice
19          | Some st –> (ctxt, Choice (c′, new_procs), st, vars)
20          end
21        end
22    | wrong –> error (ChannelHasWrongSType (c′, wrong))
23    end
24  | None –> error (NoSuchChannelInContext c′)
25  end
26  end
```

The synthesis function above matches `process` to a process offering a choice session. If the choice is on the process' own channel c, then we are dealing with an external choice, meaning the process will have its behavior dictated by another (line 3).the function will then call `synth_external_choice_proc_list`, which types all the label-process pairs in the choice, returning the output context, the the label-type pairs resulting from synthesis, the annotated process pairs, and a list of used variables for use in the compilation phase (lines 4-5). After making sure the choice is not empty, meaning there are label-process pairs in the choice, the function returns the output context, The annotated choice process and its type, as well as a list of used variables (lines 6-9). If, on the other hand, the choice refers to a process that exists in the linear context, then the function will first unfold the process' type, in case it is the recursive session type (line 11). It will then call `synth_internal_choice_proc_list`, which types the internal choice's label pairs, while at the same type ensuring the choice has sufficient label-process pairs to match the internal choice session. `synth_internal_choice_proc_list` returns the output context, the type in the label-process pairs, the annotated processes, and a list of used variables (lines 13-14).

Custom Types It is during the type checking phase that custom types are replaced in the abstract syntax tree by their most primitive types.

4.2 Compilation

We chose to compile our language to Go in order to take advantage of Go's efficient, channel-based concurrency primitives and lightweight threads (*goroutines*), instead of having to deploy our own, necessarily less efficient concurrent runtime. Moreover, by compiling to a high-level language that is not dissimilar from our source language we can leverage the compiler engineering put in place in the Go compiler rather than having to deploy such techniques from scratch.

The compilation of functions and processes is mostly straightforward. A process value in our language of the form $c \leftarrow \{P\} \leftarrow \overline{d}$ is compiled into a Go function which takes as argument the channels $\overline{d}$ and c. Channel forwarding compilation is non-trivial and detailed in Section 4.2.2. Functions are compiled straightforwardly to Go functions.

Our design choice requires addressing the differences in the typing discipline for channels between Go and our language. In Go, a channel's type codifies the type of values that can be exchanged over the channel, with this type being fixed at channel creation. In our session-typed language, a channel's payload changes over time as the session protocol is carried out. For example, in the term `send` c `4 ;send` c `true ;close` c, where c has type $int \wedge bool \wedge \mathbf{1}$, the payload of channel c is at first an integer, followed by a boolean value, and a termination message. This kind of pattern cannot be implemented directly using a simple channel type in Go. To solve this issue, we take advantage of the type information that is collected and integrated into the abstract syntax tree during the type checking stage and encode a single session channel using multiple Go channels, inspired by the translation of session types into linear types [6]. More precisely, we use a different Go channel per session *operation*, such that each operation involves not only sending the specified payload but also the *channel* on which the next action of the session protocol will take place.

Encoding Session Channels in Go. Since session types are inherently stateful objects, where each communication action advances the session to a next state (e.g. communicating an integer over a session channel of type $int \wedge bool \wedge \mathbf{1}$ advances the session type to $bool \wedge \mathbf{1}$, where a value of type $bool$ must be exchanged), we encode each session type that is used in a program into a set of protocols or session type states which are represented using custom Go types, associating to those types methods that implement the appropriate communication actions. While not all session types used in a program are explicitly provided by the user, this information is reconstructed and added to the program's abstract syntax tree during type checking.

In the case of a data exchange, the custom type contains information about the channel name, the type of the exchanged data, and information about the next state in the state machine. In the case of a choice session type, instead of just information about the next type, the correspondent custom type contains a mapping of labels to their respective next states. For a terminal session type, there is no next type, since communication ends.

Associated to each type are transition methods that advance the session, as well as the initialization functions of the types. The following is a snippet of the code generated for the type $int \wedge bool \wedge \mathbf{1}$, corresponding to the initial state (Figure 4.1 shows the full generated code; note that type `interface{}` represents any type in Go):

```go
type _send_int struct {
  c chan int
  next *_send_bool
}
func init_send_int() *_send_int {
  return &_send_int{make(chan int), nil}
}
func (x *_send_int) Send(v int) *_send_bool {
  if x.next == nil {
    x.next = init_send_bool()
  }
  x.c <- v
  return x.next
}
func (x *_send_int) Recv() (int, *_send_bool) {
    if x.next == nil {
    x.next = init_send_bool()
  }
  return <-x.c, x.next
}
```

The initial protocol state, codified by the Go struct type `_send_int`, contains a channel on which to exchange an integer value and a reference to the next protocol state, whose type is suggestively named as `_send_bool`. Since the session type specifies a value communication, `Send` and `Recv` methods are associated with the type, performing the appropriate (type-safe) communication on the underlying Go channel and returning the next state accordingly. State initialization is performed *lazily*, and so each `init` function only initializes the outer state, with the next states being initialized by the `Send` and `Recv` operations as needed. This prevents unnecessary channel creation and provides a simpler initialization in the presence of recursive types.

To illustrate the challenge of representing recursive session types, recall type `IntCStream` from Section 3.1, defined as type `rec x.&{next:int^x, stop: 1}`. Such a type codifies

```
type _send_int struct {
  c chan int
  next *_send_bool
}
func init_send_int() *_send_int {
  return &_send_int{make(chan int), nil}
}
func (x *_send_int) Send(v int) *_send_bool {
  if x.next == nil {
    x.next = init_send_bool()
  }
  x.c <- v
  return x.next
}
func (x *_send_int) Recv() (int, *_send_bool) {
    if x.next == nil {
    x.next = init_send_bool()
  }
  return <-x.c, x.next
}
type _send_bool struct {
  c chan bool
  next *_close
}
func init_send_bool() *_send_bool {
  return &_send_bool{make(chan bool), nil}
}
func (x *_send_bool) Send(v bool) *_close {
  if x.next == nil {
    x.next = init_close()
  }
  x.c <- v
  return x.next
}
func (x *_send_bool) Recv() (bool, *_close) {
    if x.next == nil {
    x.next = init_close()
  }
  return <-x.c, x.next
}
type _close struct {
  c chan interface{}
}
func init_close() *_close { return &_close{make(chan interface{})} }
func (x *_close) Send(v interface{}) { x.c <- v }
func (x *_close) Recv() interface{} { return <-x.c }
```

Figure 4.1: Compilation Preamble of *int* ∧ *bool* ∧ **1**

a protocol in which, first, a message label is exchanged. If the label is `stop`, the protocol terminates; if the label is `next`, an integer value is exchanged and the protocol repeats. As mentioned above, we compile a choice session using a message-label map, where each label is associated with an appropriately initialized representation of the corresponding branch type:

```
type _choice struct {
  c chan string
  ls map[string]interface{}
}
func init_choice() *_choice {
  m := make(map[string]interface{})
  m["stop"] = init_close()
  m["next"] = init_send_int()
  return &_choice{make(chan string), m}
}
func (x *_choice) Send(v string) { x.c <− v }
func (x *_choice) Recv() string { return <−x.c }
```

The Go type that corresponds to the `next` branch then contains an indirect reference to the choice type, allowing for the protocol to repeat: `type _send_int struct { c chan int ; next *_choice }`. Figure 4.2 shows the full preamble that is compiled for type `rec x.&{next:int^x, stop: 1}`.

The main issue with this representation arises from the fact that in Go types are treated *nominally*, whereas session types are treated *structurally*. As mentioned in Section 4.1, the session type `IntCStream` must be considered identical to `rec x.&{next:int^rec x.&{next:int^x, stop: 1}, stop: 1}` during type checking. This means that in our Go representation of sessions, we should map both `IntCStream` and all its recursive unfoldings to the Go type `_choice` above. To achieve this, we maintain during the compilation process a mapping of session types to Go types that is *quotiented* by session subtyping, effectively identifying all unfoldings of a given recursive type with the same Go type, and allowing us to reuse the same type declarations for all possible recursive usages of a recursive type. This mapping also enables an optimization in which we can reuse Go type declarations for identical session types used throughout a program.

Variable usage. While the typing for our language enforces that session channels must follow a linear usage discipline, there is no such restriction put in place to communicated functional values. A process that receives a value may simply discard it or use it an arbitrary number of times. In the former scenario, this causes an issue in our compilation to Go, where all declared variables *must* be used at least once. To address this issue, our compiler keeps track of all identifiers that are used in a given scope, replacing unused identifiers with Go's blank identifier, which may be safely ignored by the Go compiler.

Furthermore, the Go compiler does not allow for reassignment of variables with

```
type _choice struct {
  c chan string
  ls map[string]interface{}
}
func init_choice() *_choice {
  m := make(map[string]interface{})
  m["stop"] = init_close()
  m["next"] = init_send_int()
  return &_choice{make(chan string), m}
}
func (x *_choice) Send(v string) { x.c <− v }
func (x *_choice) Recv() string { return <−x.c }

type _state_1 struct {
  c chan int
  next *_choice
}
func init_send_int() *_send_int { return &_send_int{make(chan int), nil} }
func (x *_send_int) Send(v int) *_choice {
  if x.next == nil {
    x.next = init_choice()
  }
  x.c <− v
  return x.next
}
func (x *_send_int) Recv() (int, *_choice) {
  if x.next == nil {
    x.next = init_choice()
  }
  return <−x.c, x.next
}

type _close struct {
  c chan interface{}
}
func init_close() *_close { return &_close{make(chan interface{})} }
func (x *_close) Send(v interface{}) { x.c <− v }
func (x *_close) Recv() interface{} { return <−x.c }
```

Figure 4.2: Compilation Preamble for $\mu X. \& \{next : int \wedge X, stop : \mathbf{1}\}$

different types within the same scope. That is, if a variable has been assigned a value of a given type, it can not, within the current scope, ever be assigned a value of another type. This poses a problem when translating from our language into Go: in Go, the variable identifying the current state of a session has to change with every change in the session type. In the case of $int \wedge bool \wedge \mathbf{1}$, this translates into having a variable to refer to state `_send_int`, then a different variable to refer to state `_send_bool`, and a third, also different, variable to refer to the final state. To keep track of variable assigning, we maintain, during compilation, a mapping of process identifiers (identified by their channel names) to their current state identifiers (in the current scope), which are generated procedurally.

4.2.1 Compilation Phases

The compilation phase is divided into two phases, the preamble generation, and the code generation. The entry point is function `compile_prog`, which takes as argument the program to be compiled. The first step in the preamble generation phase is generating a map of session types to states, from any declarations in the program, as well from the main expression to execute. This involves creating new states from their respective session types. The structure of the states that are created is defined thusly:

```
type state_body =
  | Comm of ty * state ref
  | Choice of (string * state ref) list
  | End
and state =
  | State of string * state_body
```

A `state` is a tuple, containing the state's name/identifier, and the body of the state itself. The `state` is defined this way because often, the only part of the state that is needed for compilation is the state's name (which matches the respective compiled Go type's name). The state's body which may be of one of three types: `Comm`, which represents a send/receive session type ($\multimap, \otimes, \wedge, \supset$), and contains the type of value exchanged in the communication, and a reference to a next state; `Choice`, that represents a choice session type ($\&, \oplus$) and contains a list of associations of labels and references to their respective next states; and `End`, which simply represents the termination session type ($\mathbf{1}$). The usage of references in `Comm` and `Choice` states is justified by the need to create states involved in a cycle (due to recursion), that cannot be created unless their next state is already generated. The references allow for the creation of states with dummy references, which can be altered later, solving this problem.

The function that generates the map of type-state associations is `make_state_trees` (simplified for ease of comprehension, so no longer valid Ocaml code):

```
rec make_state_trees_from_exp map e : TypeTable(stype, state) ->
exp -> TypeTable(stype, state)
```

The above function takes as arguments: `map`, a map containing associations of session types and states; and e, an expression to be traversed. Note that in the association map, the association key is the session type; structurally equal session types are treated as the same key in the map, meaning recursive types and their respective unfoldings are treated as the same key. There is an exploration of the abstract syntax trees of both declarations and the main expression of the program, searching for process expressions, and retrieving their session types (which were put in place during typechecking). For each session type that is found, there is a call to `make_state`, which modifies the map of associations. Traversal continues, possibly altering the map further, before it is returned.

```
rec make_state map env sty : state TypeTable(stype, state) ->
(string * state ref) list -> stype -> TypeTable(stype, state)
```

The `make_state` function takes as arguments: `map`, a map of session type-state pairs; `env`, an environment containing associations of recursion variables and dummy state references (this environment keeps track of already encountered recursive types, through their respective recursion variables); and `sty`, the session type to build a state from. The function returns the passed session type-state map, enriched with the association of the received type and the newly generated state, as well as any associations related to the underlying session types. The function has a small performance optimization: if the received session type already exists in a pair in the type-state map, the map is returned immediately. Special attention should be payed to the case of creating a state from a recursive session type, which involves an indirect reference to its next state:

```
rec make_state map env sty =
  match TypeTable.find_opt map sty with
  | Some _ -> map
  | None -> begin match sty with
    | STRec (v, st) -> begin match List.assoc_opt v env with
      | Some _ -> map
      | None -> let n_state = State ("%", End) in
       let start_ref = ref (n_state) in
         let nenv = (v, start_ref)::env in
           let unfolded_st = subst_stype sty v st in
           let new_map = make_state map nenv unfolded_st in
               start_ref := TypeTable.find new_map unfolded_st;
               TypeTable.replace new_map sty !start_ref; new_map
```

Note that `STRec(v, st)` is equivalent to $\mu X.A$; v being the recursion variable X, and st the recursion body A. If the session type is a recursive type, the function checks `env` for the existence of recursion variable v. If it exists, then the recursive type has already been encountered and already exists in the association map which is promptly returned. If it does not exist, then the function will first create `n_state`, a dummy state, if you will (line 7), which will not be present in the final type-state map. Next, a reference is created to such dummy state (line 8); this will be the reference that “loops” around the states in the recursion, and starts the cycle again. The next step is adding to the environment a pair containing the recursion variable of the recursive type, and the newly created reference; this pair will signal to the final state pre-recursion that its reference must be treated in a specific way (line 9). Afterwards, there is a substitution of the recursion variable by the recursive type in the continuing session type, resulting in `unfolded_st` (line 10), ensuring the `make_state` function will keep working with closed recursive types. Then the function itself is called again for `unfolded_st` (line 11), and the reference created at the start is changed to point to the state matching `unfolded_st` (line 12), thus closing the cycle. The association map is then altered and returned.

```
 1  rec make_state map env sty =
 2    match TypeTable.find_opt map sty with
 3    | Some _ -> map
 4    | None -> begin match sty with
 5      | STSend (t, st) -> let state_id = TypeStore.get_id sty in
 6        begin match st with
 7        | STRec (v, _) -> begin match List.assoc_opt v env with
 8          | Some dummy_ref -> let this_state = State (state_id, Comm (t, dummy_ref)) in
 9            TypeTable.replace map sty this_state; map
10          | None -> let new_map = make_state map env st in
11            let this_state = State (state_id, Comm (t, ref (TypeTable.find new_map st))) in
12              TypeTable.replace new_map sty this_state; new_map
13          end
14        | _ -> let new_map = make_state map env st in
15          let this_state = State (state_id, Comm (t, ref (TypeTable.find new_map st))) in
16            TypeTable.replace new_map sty this_state; new_map
17        end
```

At the same time, take the example of the $\wedge$ session type. The function first generates a new identifier for the soon to be created state (line 5). Then a check is made; if the continuing session type is not the recursive type, the function proceeds “straightforwardly”, calls itself for the continuing state, creates the new state proper, and modifies and returns the association map (lines 14-16). Otherwise, things become somewhat more complicated. If the continuing session type is the recursive type, the function checks for the existence of the recursion variable in the environment: if it does not exist, that means the current type is not inside a recursion with that specific recursion variable, and the function proceeds as before (lines 10-12); if it does exist, the function will create the state matching the current session type, with a reference that points to a dummy state, which will, at a later point in the execution, once the current execution returns, be changed to point to the start of the recursion (lines 8-9).

Note also the generation of the state's identifier (line 5). Identifier generation is done statically; a single integer reference is kept, from which all identifiers (which are a concatenation of a string and the former integer) draw from, incrementing the referred integer when calling for a new identifier. Since the algorithm that traverses the abstract syntax tree only explores one node at a time, it may only ever generate one state at a time. Thus, from what was previously described, the correct assignment of identifiers is guaranteed.

Preamble Text Building. Afterwards, there is a sweep of a list of all the states from the map, building a complete preamble string. The string is built up text block by text block, each matching a state. For each type of state, there is a type of text block, with a template that must be completed with information from the state itself. Below are the functions that take the different states and return the filled out text blocks:

```
let make_from_comm_template_single_channel name ty next =
  "type " ^ name ^ " struct {
    c chan " ^ "interface{}" ^ "
    next *" ^ next ^ "
  }
  func init" ^ name ^ "(c chan interface{}) *" ^ name ^ " { return &" ^ name ^ "{ c, nil } }

  func (x *" ^ name ^ ") Send(v " ^ ty ^ ") *" ^ next ^ " { if x.next == nil { x.next = init" ^ next ^
  "(x.c) }; x.c <- v; return " ^ (if name != next then "x.next" else "x") ^ "}

  func (x *" ^ name ^ ") Recv() (" ^ ty ^ ", *" ^ next ^ ") { if x.next == nil { x.next = init" ^ next ^
  "(x.c) }; return (<-x.c).(" ^ ty ^ ")," ^ (if name != next then "x.next" else "x") ^ "}
  "

let make_from_end_template_single_channel name =
  "type " ^ name ^ " struct {
    c chan interface{}
  }
  func init" ^ name ^ "(c chan interface{}) *" ^ name ^ " { return &" ^ name ^ "{ c } }
  func (x *" ^ name ^ ") Send(v interface{}) { x.c <- v }
  func (x *" ^ name ^ ") Recv() interface{} { return <-x.c }
  "

let rec make_from_label_pairs_single_channel label_pairs =
  match label_pairs with
  | (l, rf)::tail -> begin match !rf with
                          | State (next_id, _) -> "\tm[\"" ^ l ^ "\"] = init" ^ next_id ^ "( c )\n" ^
                          (make_from_label_pairs_single_channel tail)
                        end
  | [] -> ""

let make_from_choice_template_single_channel name label_pairs =
  "type " ^ name ^ " struct {
    c chan interface{}
    ls map[string]interface{}
  }
  func init" ^ name ^ "(c chan interface{}) *" ^ name ^ " { m := make(map[string]interface{})\n" ^
  (make_from_label_pairs_single_channel label_pairs) ^ "\treturn &" ^ name ^ "{ c, m } }

  func (x *" ^ name ^ ") Send(v string) { x.c <- v }
  func (x *" ^ name ^ ") Recv() string { return (<-x.c).(string) }
  "
```

As is possible to see above, the different text blocks are generated by taking the properties of their respective state types. Note, that all the text block generation functions take as argument `name`, the name/identifier of the state, which gives the name to the respective compiled Go type.

Following the preamble generation, comes a phase of compiling both the declarations and the main expression of the program. Declarations can be either functions or type definitions, so the compilation result of a declaration is either a string of a Go function, or an empty string. To effectively compile a function declaration, there must be a way to compile the function's arguments types and return type, and also a way to compile the function body.

```
rec compile_type type_map typ : TypeTable(stype, state) -> ty -> string
```

The above function is quite straightforward; it takes the type-state map and the type to actually compile and returns a string version of the type in Go code.

```
rec compile_return_exp type_map ex env : TypeTable(stype, state) ->
exp -> StrMap(string, exp) -> string

rec compile_exp type_map ex env : TypeTable(stype, state) ->
exp -> StrMap(string, exp) -> string
```

The first of the above functions, `compile_return_exp`, compiles a function body. As its name implies, it compiles an expression meant to be returned, so the resulting string is prepended by the word "return". This is the only distinction of this function from the similarly named `compile_exp`, used to compile the main program expression, and which is in everything equal to the former function otherwise. The functions take as arguments the type-state map, the expression to compile, and a functional environment of identifiers and respective functional values, which may be needed to evaluate and simplify a function application before compiling it.

```
rec compile_proc type_map p id_map env : TypeTable(stype, state) ->
proc -> SeqMap(string, int ref) -> StrMap(string, exp) -> string
```

The `compile_proc` function compiles a process, as its name implies. It takes as arguments: `type_map`, the type-state mapping; p, the process to compile; `id_map`, a mapping of process identifiers to current state identifiers; and `env`, a functional environment. We show some parts of the `compile_proc`:

```
rec compile_proc type_map p id_map env =
match p with
  | Send (d, ex, opt, prc) -> let curr_id = curr_seq_id d id_map in
    let (next_id, next_map) = next_seq_id d id_map in
        next_id ^ " := " ^ curr_id ^ ".Send(" ^ compile_exp type_map ex env ^ ")\n" ^
        compile_proc type_map prc next_map env
```

Care is taken to first obtain the actual state identifier in the current scope matching the process's channel name (line 3). Then the next identifier in sequence is obtained, as well as a new identifier map, with the new identifier association (line 4). The compilation result is the assignment to `next_id` of the state that is returned by `curr_id`'s send operation of the expression `ex`'s compilation, appended with the compilation of the continuation process (lines 5-6).

```
1   rec compile_proc type_map p id_map env =
2   match p with
3     | Spawn (d, ex, opt, prc, args) -> begin match opt with
4       | Some st -> begin match TypeTable.find_opt type_map st with
5         | Some State (id, _) -> if List.length args = 0 then
6           d ^ " := init" ^ id ^ "()\n" ^ "go " ^ compile_exp type_map ex env ^ "(" ^ d ^ ")\n" ^
7           compile_proc type_map prc id_map env
8         else
9           d ^ " := init" ^ id ^ "()\n" ^ "go " ^ compile_exp type_map ex env ^ "(" ^ d ^ ", " ^
10          compile_var_list_to_fun_args args id_map ^ ")\n" ^
11          compile_proc type_map prc id_map env
12        | None -> assert false
13        end
14      | None -> assert false
15      end
```

The function above starts by finding the state matching the session type of the spawned process (line 4). The result is the call to the respective state's initialization function, followed by a goroutine call to the spawn's compiled expression, appended by the compilation of the continuation process. There is a small difference based on the existence of arguments to be passed to the spawning (lines 6-7 and 9-11).

4.2.2 Forwarding

The compilation of the channel forwarding construct `fwd` d c is directed by the (equal) session type of channels c and d. For instance, a forwarder between an ambient channel d of type $\mathit{int} \wedge \mathbf{1}$ and an offered channel c of the same type, typed as $d{:}\mathit{int} \wedge \mathbf{1} \vdash \texttt{fwd}\ d\ c :: c{:}\mathit{int} \wedge \mathbf{1}$, must necessarily receive from d an integer and send that integer along c, wait for the termination of d and then terminate the session on c. Thus, the compilation of the forwarder primitive synthesizes a process expression by analyzing the forwarded type. This process redirects messages from the ambient channel to the offered channel, and vice-versa (e.g. if the forwarded type is an input, the forwarder first receives on c and then sends along d).

```
rec compile_fwd_stype ?start_c ?start_d sty c d type_map id_map vars :
?start_c:string –> ?start_d:string –> stype –> string –> string –>
TypeTable(stype, state) –> SeqMap(string, int ref) –> string list –> string
```

The `compile_fwd_stype` function compiles, from the session type of the process in context, *d* in the example above, the appropriate forwarding behavior between processes. It takes as arguments: the optional labeled arguments of starting *c* and *d*, where `start_c` is the identifier of the process that offers the session, and *start_d* is the identifier of the process of the same type in the linear context (these starting identifiers are needed in the case of recursion, to correctly set process identifiers, when restarting the recursive cycle); the session type of the process *d* in context; the type of *d*, *c* and *d*, the process offering the session, and the process in context, respectively; the type-state associations; the process-identifier map; and a list of recursion variables used to identify already encountered recursive processes. To better illustrate the inner workings of the `compile_fwd_stype` function, we describe some parts of the function:

```
1  rec compile_fwd_stype ?start_c ?start_d sty c d type_map id_map vars =
2  match sty with
3    | STSend (t, st) –> let curr_d = curr_seq_id d id_map in
4      let next_d, fst_map = next_seq_id d id_map in
5        let curr_c = curr_seq_id c fst_map in
6          let next_c, snd_map = next_seq_id c fst_map in
7            let aux_id = curr_c ^ curr_d in
8             let mid_d = (curr_c ^ "_" ^ curr_d) in
9              begin match st with
10              | STRec (v, _) –> begin match List.find_opt (fun el –> el = v) vars with
11                | Some _ –> aux_id ^ ", " ^ mid_d ^ " := " ^ curr_d ^ ".Recv()\n" ^
12                             start_d ^ " = " ^ mid_d ^ "\n" ^
13                             start_c ^ " = " ^ curr_c ^ ".Send(" ^ aux_id ^")\n"
14                | None –> aux_id ^ ", " ^ next_d ^ " := " ^ curr_d ^ ".Recv()\n" ^
15                           next_c ^ " := " ^ curr_c ^ ".Send(" ^ aux_id ^")\n" ^
16                           compile_fwd_stype ~start_c ~start_d st c d type_map snd_map vars
17                end
18              | _ –> aux_id ^ ", " ^ next_d ^ " := " ^ curr_d ^ ".Recv()\n" ^
19                    next_c ^ " := " ^ curr_c ^ ".Send(" ^ aux_id ^")\n" ^
20                    compile_fwd_stype ~start_c ~start_d st c d type_map snd_map vars
21             end
```

The function starts by obtaining and building necessary process and value identifiers (lines 4-8). There is a single important distinction to be made, in choosing the following behavior of the function, and that is determining whether the continuation session type is a recursive type. If it is not the recursive type, the compilation result is simply a channel redirection - receiving in d, and sending in c (lines 18-20). In the case that the continuation type is the recursive type, there is a check: if the recursive type has already been encountered (its respective recursion variable exists in the `vars` environment), care must be taken to set new starting loop values for c and d, through the use of `start_c` and *start_d* (lines 11-13); otherwise the function proceeds with redirection as normal (lines 14-16).

In the event that the forwarded type is a choice, a more particular function is called, specifically to compile the list of label-type pairs:

```
rec compile_fwd_stype_list ?start_c ?start_d l c d type_map id_map vars :
?start_c:string -> ?start_d:string -> (string * stype) list -> string ->
string -> TypeTable(stype, state) -> SeqMap(string, int ref) -> string list -> string
```

The above function takes as arguments: the optional arguments `start_c` and `start_d`, necessary to set the starting values of c and d in the case of a recursion; a list of label-type pairs; c, the channel the session is offered in; d, the channel in context; the type-state map; the process-identifier association map; and an environment containing recursion variables pertaining to already encountered recursive types. This function compiles the label-type pairs into a list of matching cases (to be embedded inside a switch). The case label is the label in the aforementioned pair; the case body is the compilation of the type in the pair. A notable part of the function is the management of the process identifiers to keep them independent between cases.

If the forwarded type is a recursive type, the forwarder code is simply embedded inside an infinite loop, taking some care for correct variable usage across loop iterations and with termination breaking out of the loop:

```
rec compile_fwd_stype ?start_c ?start_d sty c d type_map id_map vars =
match sty with
  | STRec (v, st) -> begin match List.find_opt (fun el -> el = v) vars with
    | Some _ -> ""
    | None -> let curr_c = curr_seq_id c id_map in
      let curr_d = curr_seq_id d id_map in
        "for {\n" ^
        compile_fwd_stype ~start_c:curr_c ~start_d:curr_d (unfold sty) c d type_map id_map (v::vars)
        ^ "}\n"
    end
```

To start, the function checks if the type of recursion itself has already been encountered, by checking the `vars` environment for the respective recursion variable. If so, the compilation result is the empty string; otherwise the compilation result is a call to `compile_fwd_stype`, with `start_c` and `start_d` set to the identifiers of c and d in the current scope, with the unfolded session type, and with the recursion variable v added to `vars`, embedded within a for loop.

```
rec compile_fwd_stype ?start_c ?start_d sty c d type_map id_map vars =
match sty with
  | STEnd -> let curr_d = curr_seq_id d id_map in
    let curr_c = curr_seq_id c id_map in
      curr_d ^ ".Recv()\n" ^
      curr_c ^ ".Send(nil)\n" ^
      "return\n"
   end
```

This part of the function treats the type of a session that has closed communication. It redirects a session termination signal from process to another. The compilation result contains a "return", to ensure it breaks out of a for cycle, if the session termination occurs inside a recursive type.

We highlight the compilation result of the forwarding used in the definition of `augNats` from Section 3.1 (with renaming of states for readability). Note the `next` branch, where Go variables *c* and *d* are used according to the specified type and subsequently restored with the appropriate values before the next iteration. Note the `stop` branch as well, containing the return statement that breaks out of the for loop.

```
for {
  dc1 := c1.Recv()
  d.Send(dc1)
  switch dc1 {
  case "next":
    d0 := d.ls["next"].(*_state_next)
    c2 := c1.ls["next"].(*_state_next)
    c2d0, c2_d0 := d0.Recv()
    d = c2_d0
    c1 = c2.Send(c2d0)
  case "stop":
    d0 := d.ls["stop"].(*_state_close)
    c2 := c1.ls["stop"].(*_state_close)
    d0.Recv()
    c2.Send(nil)
    return
  }
}
```

Final Program. After generating the preamble, and compiling all declarations plus the main expression, we concatenate these parts, in that order, to create the final Go program.

5 Evaluation

Measuring the performance of the compiled program is important to have a complete understanding of the possibilities and limitations of the developed language. We carried out multiple performance analysis approaches.

There are two approaches to managing Go channels in Go structs resulting from type compilation: one where a new communication channel is created per compiled type, the *multi-channel* approach; and one where a single channel is reused for each and every compiled type, the *single channel* approach. Following the multi-channel approach provides an extra guarantee of type safety, from the Go compiler's standpoint, since every channel is always well typed, but implies an added overhead from creating an additional channel for each communication state, and maintaining mutual exclusion access to the channel creation operation. This is unlike the single channel approach: Go channels can only exchange values of one single type, passed at channel creation, during their lifetime; as such, this single channel must be of type `interface{}`, meaning *any* type. This approach implies constant type casting on channel communication, which is inherently less safe, from a Go standpoint (still, type safety is presumed to be guaranteed by our implementation), and could lead to increased overhead during runtime. We thought a relevant performance evaluation would be comparing two executions of the same program, each following a different channel approach. On accounts of possible performance losses at runtime due to type casting, we also proposed testing whether the communication of different types (and different casts) would produce different results from the repeated communication of the same type.

To study the performance of compiled code, we compiled a testing program; the testing program was then executed 100 times, measuring the execution time in each execution. Then we took the average of all execution times to obtain a final experimental result. The testing program appears in Figure 5.1, and the experimental results appear in Table 5.1. To test for the impact of different data type communication, a similar program was used where instead of only sending integers, the data sent changes type with each communication step.

To figure out whether the temporal difference in the obtained results are relevant

```
let main : unit -> {1} =
fun u ->
  c <- {
    d <- spawn {
      v1:int <- recv d; v2:int <- recv d; v3:int <- recv d;
      v4:int <- recv d; v5:int <- recv d; v6:int <- recv d;
      v7:int <- recv d; v8:int <- recv d; v9:int <- recv d;
      v10:int <- recv d;

      send d v1; send d v2; send d v3;
      send d v4; send d v5; send d v6;
      send d v7; send d v8; send d v9;
      send d v10;

      close d
    };
    send d 1; send d 2; send d 3;
    send d 4; send d 5; send d 6;
    send d 7; send d 8; send d 9;
    send d 10;

    v1:int <- recv d; v2:int <- recv d; v3:int <- recv d;
    v4:int <- recv d; v5:int <- recv d; v6:int <- recv d;
    v7:int <- recv d; v8:int <- recv d; v9:int <- recv d;
    v10:int <- recv d;

    wait d;
    close c
  }
end;
```

Figure 5.1: Test Program

	Same Data Type	Different Data Type
Multi-Channel	18925.91	18615.02
Single Channel	17424.55	17349.89

Table 5.1: Average Execution Times of Test Programs in Nanoseconds

Approach 1	Approach 2	Relevant
Multi-Channel Same Data Type	Multi-Channel Different Data Type	No
Single Channel Same Data Type	Single Channel Different Data Type	No
Multi-Channel Same Data Type	Single Channel Same Data Type	Yes
Multi-Channel Different Data Type	Single Channel Different Data Type	Yes

Table 5.2: Statistical Analysis Results and Relevance of Time Differences

or not we conduct an independent two-sample *Student's t-test*, accounting for possibly unequal variances [46]. We satisfy the conditions that need to be met for this kind of statistical inference: the sampling is random and independent, and the sample follows an approximately normal distribution. We can easily check, graphically, that the samples all follow an approximately normal distribution; we do this by testing for normality via *Q-Q plot*, by plotting the percentiles of the random sample data against a normal distribution. Relevant demonstrations and calculations appear in Appendix B. The results of the statistical analysis appear in Table 5.2. We can easily conclude from the results of the statistical analysis that there is no impact in performance with regards to the type of data exchanged in communication, however there is a slight performance impact when using the multi-channel approach as opposed to the single channel approach, in the order of hundreds of nanoseconds. So it does appear that the overhead from the creation of a new channel for each possible communication state is slightly significant, moreso than any possible overhead from type casting.

One other interesting evaluation exercise would be to write a program in our language (following the single channel approach, which has been determined to be the best performing), and then write that very same program in native Go. Both programs would be executed and their performances compared. In particular, it would be interesting to check if the combination of `spawn` plus the `fwd` construct, which redirects communication from a newly spawned process to the current process and vice-versa, when used in a recursive context, impacts performance compared to the equivalent native method of simply passing the same channel to a new goroutine and terminating.

The test program in our language appears in Figure 5.2 (based of the program that appears in Section 3.1), and the test program in native Go appears in Figure 5.3. Both programs spawn a process communicating on a given channel *d* that first receives a “next” label indicating its continuation, and then return an integer, finally ending by spawning a new process that repeats the process for the next integer in sequence, communicating on the same channel *d*. After 100 cycles of sending the “next” label and receiving an integer, the `main` function sends the “stop” label, causing the termination of the outstanding concurrent processes. Each program runs 100 times, execution times are registered and the average of those 100 execution times is taken. The experimental results appear in Table 5.3.

It's plain to see, without statistical analysis, that the values differ by *two orders of*

```
stype IntCStream rec x.&{next: int^x, stop: @};
let augNats : int –> {IntCStream} =
fun n –>
  c <– {
    case c of
    next:(
      send c n;
      d <– spawn (augNats (n+1));
      fwd d c
    ) stop:(
      close c
    )
  }
end;
let main : unit –> {1} =
fun u –>
  c <– {
    d <– spawn (augNats 0);
    d.next;
    a0:int <– recv d;
    ...
    a99:int <– recv d;
    d.stop;
    wait d;
    close c
  }
end;
```

Figure 5.2: Test Program in Our Language

	Time (ns)
Our Language	4713365
Native Go	86267

Table 5.3: Execution Times of Our Language and Native Go

```go
func AugNats(c chan interface{}, n int) {
  label := <-c
  switch label.(string) {
  case "next":
    c <- n
    go AugNats(c, n+1)
    return
  case "stop":
    return
  }
}

func main() {
  d := make(chan interface{})
  go AugNats(d, 0)

  for i := 0; i < 100; i++ {
    d <- "next"
    (<-d)
  }
  d <- "stop"
}
```

Figure 5.3: Test Program in Native Go

magnitude. The performance of our language vs native Go is severely impacted by the `spawn` plus `fwd` construct's functioning.

We can conclude, overall, that with respect to the channel approach, it is marginally better to follow the single channel approach, since any possible overhead from type casting is largely irrelevant when compared to the channel creation overhead from the multi-channel approach. We can also conclude that the combination of `spawn` plus `fwd` has a large impact on the performance of the compiled code, when compared with its natively written equivalent. Indeed, since the combination is mostly used to model recursion, as is the case in the test program, there is always a necessary performance cost from spawning a new process, besides keeping the current process alive, each time the recursive cycle starts anew. A possible performance optimization would be to alter the `fwd` recursive pattern to run in the same goroutine, instead of spawning a new one at each step, along with a new communication channel.

6 Conclusion

In this work we have developed: a simple language interpreter; a complete language type checker which makes use of a bidirectional type checking technique that allows us to omit most type annotations, and gives us various safety assurances; and a full-fledged compiler that compiles programs written in our language to valid, and equivalent, Go language programs, through an *encoding* of session types to equivalent Go type states. We have also performed a relevant performance evaluation of our implementation, and produced a critical examination of the experimental results.

Our language provides, by way of its type checker, two key, compile-time, guarantees: the absence of communication errors, and deadlock-freedom in correct programs.

As for future work, we would focus on three key points: expanding the number of programs admitted by the language's type checker by, as an example, admitting the parallel composition of two well-typed processes, as exemplified in Section 3.2; defining more expressive language types, by, for instance, introducing restrictions on the values of the communicated data (such as bounding an integer); optimizing the compilation process, by packing several messages into one communication payload to minimize number of messages, or by optimizing the `fwd` pattern to use one single goroutine, to illustrate a few possibilities.

Bibliography

[1] B. Almeida, A. Mordido, and V. T. Vasconcelos. “FreeST: Context-free Session Types in a Functional Language”. In: *PLACES@ETAPS*. Vol. 291. EPTCS. 2019, pp. 12–23 (cit. on pp. 2, 13).

[2] G. R. Andrews. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991. ISBN: 978-0-8053-0086-4 (cit. on p. 5).

[3] L. Caires and F. Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *CONCUR*. Vol. 6269. Lecture Notes in Computer Science. Springer, 2010, pp. 222–236 (cit. on pp. 10, 11, 14).

[4] I. Cervesato, J. S. Hodas, and F. Pfenning. “Efficient resource management for linear logic proof search”. In: *Theor. Comput. Sci.* 232.1-2 (2000), pp. 133–163 (cit. on p. 27).

[5] R. Chen, S. Balzer, and B. Toninho. “Ferrite: A Judgmental Embedding of Session Types in Rust”. In: *ECOOP*. to appear. 2022 (cit. on p. 12).

[6] O. Dardha, E. Giachino, and D. Sangiorgi. “Session types revisited”. In: *PPDP*. ACM, 2012, pp. 139–150 (cit. on p. 35).

[7] A. Das, J. Hoffmann, and F. Pfenning. “Parallel complexity analysis with temporal session types”. In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 91:1–91:30 (cit. on p. 13).

[8] A. Das and F. Pfenning. “Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description)”. In: *FSCD*. Vol. 167. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 33:1–33:17 (cit. on p. 13).

[9] A. Das and F. Pfenning. “Session Types with Arithmetic Refinements”. In: *CONCUR*. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 13:1–13:18 (cit. on p. 13).

[10] J. Dunfield and N. Krishnaswami. “Bidirectional Typing”. In: *ACM Comput. Surv.* 54.5 (2021), 98:1–98:38 (cit. on p. 27).

[11] J. Franco and V. T. Vasconcelos. “A Concurrent Programming Language with Refined Session Types”. In: *SEFM Workshops*. Vol. 8368. Lecture Notes in Computer Science. Springer, 2013, pp. 15–28 (cit. on p. 13).

[12] S. J. Gay and M. Hole. "Subtyping for session types in the pi calculus". In: *Acta Informatica* 42.2-3 (2005), pp. 191–225. DOI: `10.1007/s00236-005-0177-z`. URL: `https://doi.org/10.1007/s00236-005-0177-z` (cit. on pp. 28, 29).

[13] S. J. Gay and M. Hole. "Types and Subtypes for Client-Server Interactions". In: *ESOP*. Vol. 1576. Lecture Notes in Computer Science. Springer, 1999, pp. 74–90 (cit. on p. 28).

[14] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *ESOP*. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 122–138 (cit. on pp. 2, 5, 6, 10, 13).

[15] K. Honda, N. Yoshida, and M. Carbone. "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 2008, pp. 273–284. DOI: `10.1145/1328438.1328472`. URL: `https://doi.org/10.1145/1328438.1328472` (cit. on pp. 8, 10).

[16] R. Hu and N. Yoshida. "Explicit Connection Actions in Multiparty Session Types". In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 2017, pp. 116–133. DOI: `10.1007/978-3-662-54494-5\_7`. URL: `https://doi.org/10.1007/978-3-662-54494-5\_7` (cit. on pp. 2, 12).

[17] K. Imai, N. Yoshida, and S. Yuen. "Session-ocaml: A session-based library with polarities and lenses". In: *Sci. Comput. Program.* 172 (2019), pp. 135–159. DOI: `10.1016/j.scico.2018.08.005`. URL: `https://doi.org/10.1016/j.scico.2018.08.005` (cit. on p. 12).

[18] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. "Session types for Rust". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*. Ed. by P. Bahr and S. Erdweg. ACM, 2015, pp. 13–22. DOI: `10.1145/2808098.2808100`. URL: `https://doi.org/10.1145/2808098.2808100` (cit. on p. 12).

[19] N. Kobayashi. "A Partially Deadlock-Free Typed Process Calculus". In: *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*. 1997, pp. 128–139. DOI: `10.1109/LICS.1997.614941`. URL: `https://doi.org/10.1109/LICS.1997.614941` (cit. on p. 10).

[20] N. Kobayashi. "A Type System for Lock-Free Processes". In: *Inf. Comput.* 177.2 (2002), pp. 122–159. DOI: `10.1006/inco.2002.3171`. URL: `https://doi.org/10.1006/inco.2002.3171` (cit. on p. 10).

[21] N. Kobayashi and D. Sangiorgi. "A Hybrid Type System for Lock-Freedom of Mobile Processes". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. 2008, pp. 80–93. DOI: 10.1007/978-3-540-70545-1_10. URL: https://doi.org/10.1007/978-3-540-70545-1_10 (cit. on p. 10).

[22] W. Kokke. "Rusty Variation: Deadlock-free Sessions with Failure in Rust". In: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*. Ed. by M. Bartoletti et al. Vol. 304. EPTCS. 2019, pp. 48–60. DOI: 10.4204/EPTCS.304.4. URL: https://doi.org/10.4204/EPTCS.304.4 (cit. on p. 12).

[23] W. Kokke and O. Dardha. "Deadlock-free session types in linear Haskell". In: *Haskell*. ACM, 2021, pp. 1–13 (cit. on p. 12).

[24] N. Lagaillardie, R. Neykova, and N. Yoshida. "Implementing Multiparty Session Types in Rust". In: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Ed. by S. Bliudze and L. Bocchi. Vol. 12134. Lecture Notes in Computer Science. Springer, 2020, pp. 127–136. DOI: 10.1007/978-3-030-50029-0_8. URL: https://doi.org/10.1007/978-3-030-50029-0_8 (cit. on p. 12).

[25] S. Lindley and J. G. Morris. "Embedding session types in Haskell". In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. 2016, pp. 133–145. DOI: 10.1145/2976002.2976018. URL: https://doi.org/10.1145/2976002.2976018 (cit. on p. 12).

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

[27] M. Neubauer and P. Thiemann. "An Implementation of Session Types". In: *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*. 2004, pp. 56–70. DOI: 10.1007/978-3-540-24836-1_5. URL: https://doi.org/10.1007/978-3-540-24836-1_5 (cit. on p. 12).

[28] R. Neykova and N. Yoshida. "Multiparty Session Actors". In: *Logical Methods in Computer Science* 13.1 (2017). DOI: 10.23638/LMCS-13(1:17)2017. URL: https://doi.org/10.23638/LMCS-13(1:17)2017 (cit. on p. 12).

[29] R. Neykova et al. "A session type provider: compile-time API generation of distributed protocols with refinements in F#". In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 2018, pp. 128–138. DOI: 10.1145/3178372.3179495. URL: https://doi.org/10.1145/3178372.3179495 (cit. on p. 12).

[30] D. A. Orchard and N. Yoshida. "Effects as sessions, sessions as effects". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 568–581. DOI: 10.1145/2837614.2837634. URL: https://doi.org/10.1145/2837614.2837634 (cit. on p. 12).

[31] L. Padovani. "A simple library implementation of binary sessions". In: *J. Funct. Program.* 27 (2017), e4. DOI: 10.1017/S0956796816000289. URL: https://doi.org/10.1017/S0956796816000289 (cit. on p. 12).

[32] J. A. Pérez et al. "Linear Logical Relations for Session-Based Concurrency". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 539–558. DOI: 10.1007/978-3-642-28869-2_27. URL: https://doi.org/10.1007/978-3-642-28869-2_27 (cit. on pp. 10, 11).

[33] F. Pfenning and D. Griffith. "Polarized Substructural Session Types". In: *FoSSaCS*. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 3–22 (cit. on pp. 13, 14).

[34] B. C. Pierce and D. N. Turner. "Local Type Inference". In: *POPL*. ACM, 1998, pp. 252–265 (cit. on p. 27).

[35] R. Pucella and J. A. Tov. "Haskell session types with (almost) no class". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 2008, pp. 25–36. DOI: 10.1145/1411286.1411290. URL: https://doi.org/10.1145/1411286.1411290 (cit. on p. 12).

[36] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001. ISBN: 978-0-521-78177-0 (cit. on p. 10).

[37] A. Scalas and N. Yoshida. "Lightweight Session Programming in Scala". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 21:1–21:28. DOI: 10.4230/LIPIcs.ECOOP.2016.21. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.21 (cit. on p. 12).

[38] A. Scalas and N. Yoshida. "Multiparty session types, beyond duality". In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 55–84. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2018.01.001. URL: https://www.sciencedirect.com/science/article/pii/S2352220817301487 (cit. on p. 10).

[39] A. Scalas, N. Yoshida, and E. Benussi. “Effpi: verified message-passing programs in Dotty”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*. 2019, pp. 27–31. DOI: `10.1145/3337932.3338812`. URL: `https://doi.org/10.1145/3337932.3338812` (cit. on p. 12).

[40] A. Scalas et al. “A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming”. In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by P. Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 24:1–24:31. DOI: `10.4230/LIPIcs.ECOOP.2017.24`. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2017.24` (cit. on p. 12).

[41] B. Toninho, L. Caires, and F. Pfenning. “Dependent session types via intuitionistic linear type theory”. In: *PPDP*. ACM, 2011, pp. 161–172 (cit. on p. 14).

[42] B. Toninho, L. Caires, and F. Pfenning. “Higher-Order Processes, Functions, and Sessions: A Monadic Integration”. In: *ESOP*. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 350–369 (cit. on pp. 2, 13, 14, 17, 22, 62).

[43] A. L. Voinea, O. Dardha, and S. J. Gay. “Typechecking Java Protocols with [St]Mungo”. In: *Formal Techniques for Distributed Objects, Components, and Systems*. Springer International Publishing, 2020, pp. 208–224. DOI: `10.1007/978-3-030-50086-3_12`. URL: `https://doi.org/10.1007%2F978-3-030-50086-3_12` (cit. on p. 12).

[44] A. L. Voinea, O. Dardha, and S. J. Gay. “Typechecking Java Protocols with [St]Mungo”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Ed. by A. Gotsman and A. Sokolova. Vol. 12136. Lecture Notes in Computer Science. Springer, 2020, pp. 208–224. DOI: `10.1007/978-3-030-50086-3\_12`. URL: `https://doi.org/10.1007/978-3-030-50086-3\_12` (cit. on pp. 2, 12).

[45] P. Wadler. “The Marriage of Effects and Monads”. In: *ICFP*. ACM, 1998, pp. 63–74 (cit. on p. 14).

[46] B. L. Welch. “The generalisation of student’s problems when several different population variances are involved.” In: *Biometrika* 34 1-2 (1947), pp. 28–35 (cit. on pp. 52, 66).

[47] M. Willsey, R. Prabhu, and F. Pfenning. “Design and Implementation of Concurrent C0”. In: *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016*. 2016, pp. 73–82. DOI: `10.4204/EPTCS.238.8`. URL: `https://doi.org/10.4204/EPTCS.238.8` (cit. on p. 13).

[48] M. Willsey, R. Prabhu, and F. Pfenning. “Design and Implementation of Concurrent C0”. In: *Electronic Proceedings in Theoretical Computer Science* 238 (2017), pp. 73–82. DOI: 10.4204/eptcs.238.8. URL: https://doi.org/10.4204%2Feptcs.238.8 (cit. on p. 62).

A

Examples

We show a number of examples that showcase the expressiveness of our language.

First, a queue implementation where each element is a concurrent process (found in [48]), as can be seen in Figure A.1. In this queue, the process of enqueuing an integer leads to the successive passing of that integer from the element at the head of the queue all the way to the tail of the queue. This is done, of course, concurrently, meaning it's possible to enqueue a given element while others are still being transmitted to the back of the queue.

Similarly to the concurrent queue shown in Figure A.1, is the example of the concurrent list implementation in Figure A.2 (found in [42]). Each process represents an element of the list.

Finally, as another example of a concurrent data structure is a concurrent stack (found in [42]), which appears in in Figure A.3. The concurrent stack makes use of the concurrent list already defined in Figure A.2.

```
stype Queue = rec x. &{enq: int => x, deq: +{some: int ^ x, none: 1}};

let elem : int –> {Queue <–r:Queue} =
fun x –>
  q <– {
    case q of
    enq:(
      y:int <– recv q; r.enq; send r y;
      q_ <– spawn (elem x) r; fwd q_ q
    )
    deq:( q.some; send q x; fwd r q )
  }
end;

let empty : unit –> {Queue} =
fun u –>
  q <– {
    case q of
    enq:(
      y:int <– recv q; print y;
      e <– spawn (empty ());
      q_ <– spawn (elem y) e; fwd q_ q
    )
    deq:( q.none; close q )
  }
end;

let dealloc : unit –> {1 <–q:Queue} =
fun u –>
  c <– {
    q.deq;
    case q of
    some:( i:int <– recv q; d <– spawn (dealloc ()) q; fwd d c )
    none:( wait q; close c )
  }
end;

let main : unit –> {1} =
fun u –>
  c <– {
    q <– spawn (empty ());
    q.enq; send q 1; q.enq; send q 2;
    d <– spawn (dealloc ()) q;
    wait d;
    close c
  }
end
```

Figure A.1: Concurrent Queue

```
let stype IntList = rec x. +{null: 1, cons: int ^ (x) ∗ 1};
let null : unit −> {IntList} =
fun n −>
  c <− {
    c.null;
    close c
  }
end;
let cons : int −> {IntList <−l:IntList} =
fun v −>
  c <− {
    c.cons;
    send c v;
    l_ <− spawn { fwd l l_ } l;
    sendc c l_;
    close c
  }
end;
let dealloc : unit −> {1 <−l:IntList} =
fun v −>
c <− {
  case l of
  null:(
    wait l;
    close c
  )
  cons:(
    n:int <− recv l;
    l_:IntList <− recvc l;
    wait l;
    d <− spawn (dealloc ()) l_;
    fwd d c
  )
 }
end;
let main : unit −> {1} =
fun u −>
  c <− {
    e0 <− spawn (null ());
    e1 <− spawn (cons 1) e0;
    e2 <− spawn (cons 2) e1;
    e3 <− spawn (cons 3) e2;
    e4 <− spawn (dealloc ()) e3;
    wait e4;
    close c
  }
end;
```

Figure A.2: Concurrent List

```
let stype IntStack = rec x. &{push: int => x, pop: +{none: unit ^ x, some: int ^ x}, dealloc: 1};
let stack : unit -> {IntStack <-l:IntList} =
fun n ->
   c <- {
      case c of
      push: ( v:int <- recv c; l_ <- spawn (cons v) l; d <- spawn (stack ()) l_; fwd d c )
      pop: (
         case l of
         null: ( wait l; c.none; send c (); l_ <- spawn (null ()); d <- spawn (stack ()) l_; fwd d c )
         cons: (
            v:int <- recv l; l_:IntList <- recvc l; wait l; c.some; send c v;
            d <- spawn (stack ()) l_; fwd d c
            )
      )
      dealloc: ( d <- spawn (dealloc ()) l; fwd d c )
   }
end;
let main : unit -> {1} =
fun u ->
  c <- {
   l <- spawn (null ()); s <- spawn (stack ()) l;
   s.push; send s 1; s.push; send s 2;
   s.pop;
   case s of
   none:(
      v:unit <- recv s;
      s.dealloc; wait s;
      close c
   )
   some:(
      u:int <- recv s;
      s.dealloc; wait s;
      close c
   )
  }
end;
```

Figure A.3: Concurrent Stack

B

Results

To verify if the differences between the obtained results are statistically relevant we conduct a *Student's t-test*, more specifically *Welch's t-test* variant [46], which does not assume equal variance. We are at liberty to do this kind of inference: the sampling is random and independent; and the sample follows an approximately normal distribution, which can be surmised by observing Figure B.1, Figure B.2, Figure B.3, and Figure B.4, which feature *Q-Q plots* of the 100 data points for each experimental result plotted against a normal distribution. As evidenced by the clear overlapping of the data points with the trendlines, we can determine, graphically, that the samples were pulled from a population which follows an approximately normal distribution (the samples were all pulled from the same population, so to speak, so it tracks that they would all follow the same distribution).

On to the t-test itself; we conduct the test for each temporal difference between related

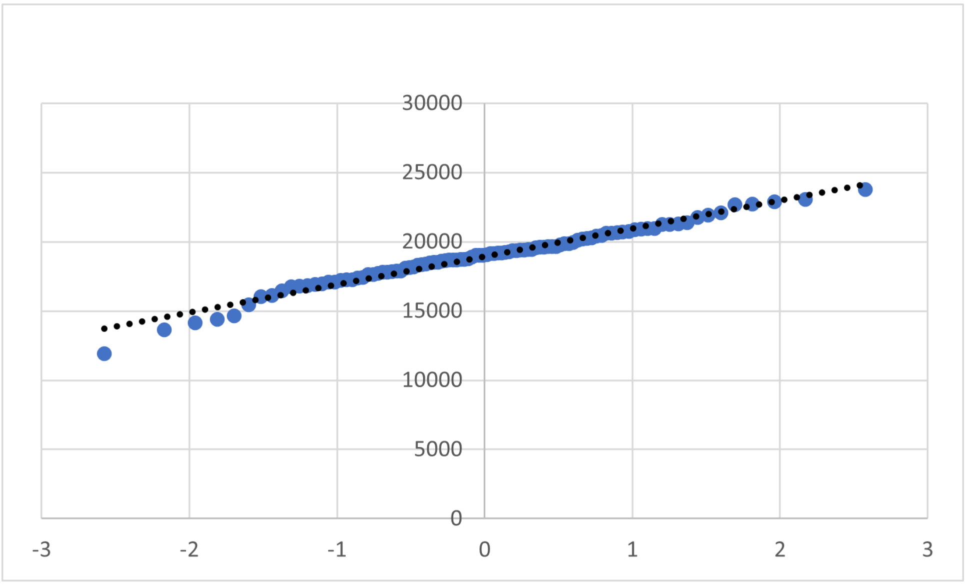

Figure B.1: Q-Q Plot: Multi-Channel Same Data Type

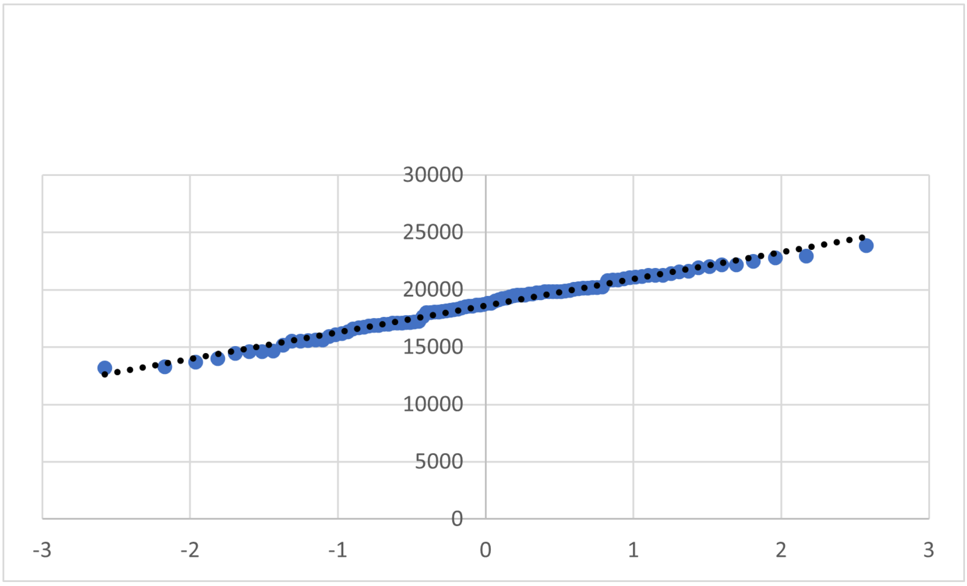

Figure B.2: Q-Q Plot: Multi-Channel Different Data Type

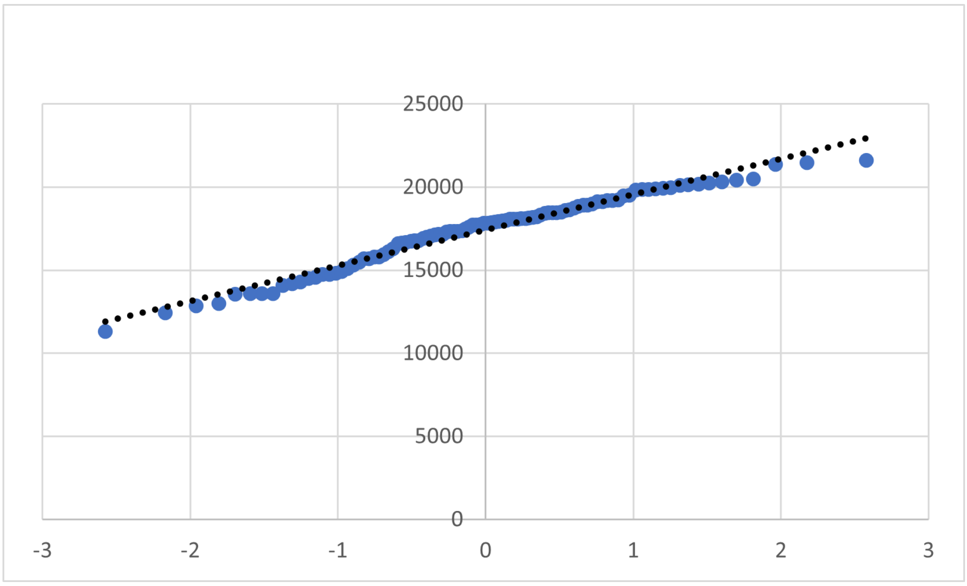

Figure B.3: Q-Q Plot: Single Channel Same Data Type

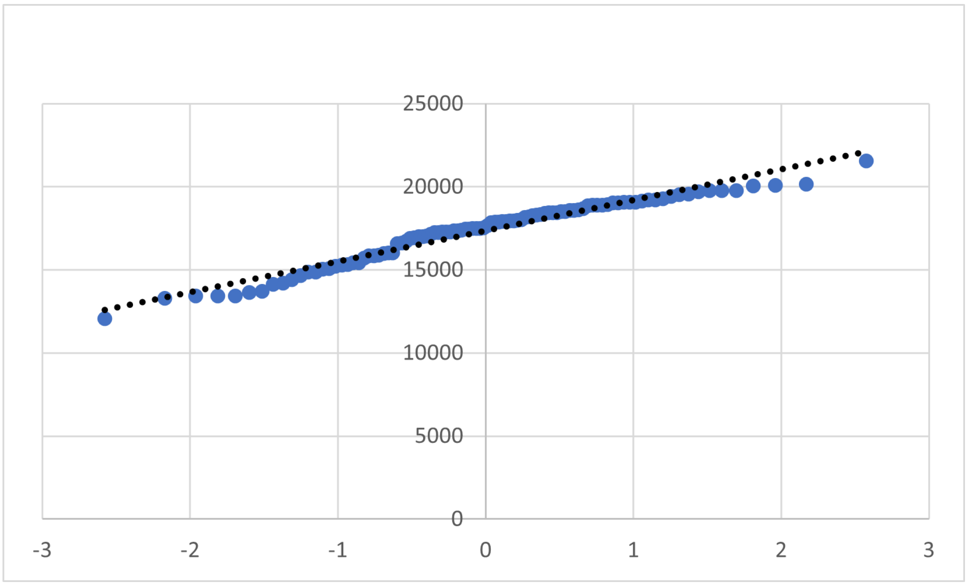

Figure B.4: Q-Q Plot: Single Channel Different Data Type

	Mean (ns)	StdDev (ns)	n
Multi-Channel Same Data Type	18925.91	2044.97	100
Multi-Channel Different Data Type	18615.02	2344.07	100
Single Channel Same Data Type	17424.55	2163.58	100
Single Channel Different Data Type	17349.89	1887.91	100

Table B.1: Relevant Statistics of Obtained Results

experimental results. We show the work for the example of the temporal difference between `Multi-Channel Same Data Type` and `Multi-Channel Different Data Type`, and simply show the results for the remaining cases. Relevant statistics are displayed in Table B.1.

Be it that our null hypothesis H_0 is that the means (obtained results) are equal, i.e. there is no relevant temporal difference between the two of them; and be it that H_1 is that the means are not equal. Let the significance level α equal 0.05 (meaning we will reject/accept a hypothesis with a 95% confidence level).

First we calculate the degrees of freedom of the test variable t, where S = StdDev and $n = 100$:

$$d.f. = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{\left(S_1^2/n_1\right)^2}{n_1 - 1} + \frac{\left(S_2^2/n_2\right)^2}{n_2 - 1}}$$

Approach 1	Approach 2	H_0	Relevant
Multi-Channel Same Data Type	Multi-Channel Different Data Type	Accepted	No
Single Channel Same Data Type	Single Channel Different Data Type	Accepted	No
Multi-Channel Same Data Type	Single Channel Same Data Type	Rejected	Yes
Multi-Channel Different Data Type	Single Channel Different Data Type	Rejected	Yes

Table B.2: Statistical Analysis Results and Relevance of Time Differences

So we have that:

$$d.f. = \frac{\left(\frac{2044.97^2}{100} + \frac{2344.07^2}{100}\right)^2}{\frac{\left(2044.97^2/100\right)^2}{100-1} + \frac{\left(2344.07^2/100\right)^2}{100-1}} \approx 194$$

For an α of 0.05 and 194 degrees of freedom, we have that the critical value, obtained from a Student's t table (two tailed since we are dealing with a normal distribution), is 1.97. As such, if the test variable t is greater than 1.97, then we reject the null hypothesis H_0, meaning the time difference would be statistically relevant.

The next step is calculating the test variable:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

And so, we have that:

$$t = \frac{18925.91 - 18615.02}{\sqrt{\frac{2044.97^2}{100} + \frac{2344.07^2}{100}}} = 1$$

Since 1 is lesser than 1.97, we accept the null hypothesis with 95% confidence. This means that the temporal difference between the two experimental results is not statistically relevant.

As previously mentioned, we now present the statistical conclusions of all the possible differences between experimental results, in Table B.2.

2022 Making Session Types Go

João Geraldo

NOVA SCHOOL OF SCIENCE & TECHNOLOGY