João Miguel Pereira do Cano Rico Geraldo

Bachelor of Science

Designing and Implementing a Compiled Programming Language with Session-Typed Concurrency

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2020

Abstract

Modern software practices fundamentally require programmers to take a concurrency-first approach to program development and design, due to heterogeneous resource interaction, strong availability requirements and explosion in ubiquitousness of multi-core architectures. With the shift towards pervasive concurrent programming, programming languages have also evolved in terms of concurrency related features, providing users with high-level concurrency primitives such as green-threads, thread-pools, mailboxes, channels, and mechanisms for asynchronous execution (e.g. async-await, events, futures), that enable programmers to express higher-level designs in a more natural way. However, even languages with strong type-safety guarantees are not able to provide any but the most basic static correctness guarantees in these concurrent settings, being only able to ensure type safety but failing to guarantee deadlock-freedom or protocol fidelity.

In this work, we leverage a family of type systems for concurrency dubbed *session types*, which can be used to statically ensure communication safety and protocol fidelity, in order to implement a concurrent programming language that provides high-level concurrency features such as communication channels, process spawning and channel passing, and also strong-safety guarantees such as protocol fidelity and deadlock-freedom. Concretely, we will develop a type checker, interpreter and compiler for a session-typed functional language with the features mentioned above, whose design is inspired by previous work. The type checker will employ state-of-the-art bidirectional type checking techniques, and the compiler will target a realistic backend (such as the JVM or LLVM), potentially making use of the logical properties of the type system to optimize execution.

Keywords: Concurrency, High-Level Concurrency Features, Session Types, Deadlock-Freedom, Type System, Logical Session Types, Linear Types, Statically Certified Safety Guarantees

Resumo

As práticas de software modernas exigem que os programadores tomem uma abordagem que privilegie a concorrência aquando do desenvolvimento e desenho de programas, devido à interacção de recursos heterogéneos, requisitos de disponibilidade rigorosos, e explosão na ubiquidade das arquitecturas *multi-core*. Com a transição para a generalização da programação concorrente, as linguagens de programação evoluíram em termos de funcionalidades relacionadas com concorrência, oferecendo aos utilizadores primitivas de concorrência de alto nível, tais como *green-threads*, *thread-pools*, *mailboxes*, canais, e mecanismos de execução assíncrona (e.g. *async-await*, eventos, futuros), que permitem aos programadores expressar desenhos de alto nível de um modo mais natural. No entanto, mesmo as linguagens com garantias robustas de segurança de tipos não conseguem assegurar senão as garantias estáticas mais simples no que toca à correcção em ambientes concorrentes, garantido apenas segurança de tipos, não sendo capazes de garantir ausência de *dealocks* ou fidelidade de protocolos.

Neste trabalho, tiramos proveito duma família de sistemas de tipos para concorrência denominados de *tipos de sessão*, que podem ser utilizados para assegurar estaticamente a segurança de comunicações e a fidelidade de protocolos, de modo a implementar uma linguagem de programação concorrente que oferece funcionalidades de concorrência de alto nível, tais como canais de comunicação, *process spawning*, e *channel passing*, e dá garantias robustas de segurança, tais como fidelidade de protocolos e ausência de *deadlocks*. Mais concretamente, vamos desenvolver um *type checker*, um interpretador e um compilador para uma linguagem funcional, com um sistema de tipos de sessão, e com as funcionalidades supra mencionadas, cujo desenho se baseia em trabalho prévio. O *type checker* aplicará as técnicas mais modernas de *type checking* bidireccional, e o compilador terá por alvo um *backend* realístico, possivelmente tirando partido das propriedades lógicas do sistema de tipos para optimizar a execução.

Palavras-chave: Concurrência, Funcionalidades de Concurrência de Alto Nível, Tipos de Sessão, Ausência de *Deadlocks*, Sistema de Tipos, Tipos de Sessão Lógicos, Tipos Lineares, Garantias de Segurança Certificadas Estaticamente

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Concurrent programming is increasingly ubiquitous - we see today's applications being distributed across several machines, subject to strong availability requirements, and forced to manage communication between various heterogeneous computational resources (e.g. web servers or authentication servers) simultaneously. With this ever growing demand for concurrency, programming languages evolved, coming to offer and increasing number of high-level features for concurrent programming such as green-threads, thread-pools, mailboxes, channels, and various different mechanisms for asynchronous execution (e.g. async-await, events, futures). Though the support for it has increased, concurrent programming remains very complex and error-prone, especially when compared to more traditional sequential programming since it's harder to reason about and assert program correctness in a concurrent setting, due to non-deterministic process interleaving and interference.

Given the pervasiveness of large-scale concurrent applications, programming errors can prove to be disastrous [25, 33, 49], and quite costly. Therefore, the safety of concurrent programming is paramount in the modern world; still, most mainstream languages offer little support for "safe" concurrent programming. Indeed, most programming languages can only provide static guarantees of basic type safety, which does not map to crucial concurrency-related properties, such as deadlock-freedom, race-freedom or protocol fidelity. As such, having little way to ensure such properties at compile-time, safety in concurrent programming is mostly "left to the responsibility" of programmers, who are, by their very nature, fallible.

1.2 Proposed Work

We propose to develop a new general-purpose concurrent language, with strong static correctness guarantees, featuring a *session type system*. Session types are a type-level way of specifying series of reciprocal interactions between processes at a *high-level* of abstraction, which can be used to represent communication patterns ranging from the simple (e.g call-return and method invocation) to the complex (e.g. continuous interactions, unbounded interactions, and delegation of processing tasks to other processes). Our goal is to design the language in a way that leverages the static correctness properties of session types (i.e. deadlock-freedom and protocol compliance), but remains syntactically and semantically intuitive for users, so as to make it an attractive option for use in concurrent programming. The proposed language will combine functional programming features, such as parametric polymorphism, higher-order functions, inductive data types, and pattern matching, with high-level message-passing concurrency features, such as session-typed communication channels, process spawning, and channel passing. The language will thus combine the static correctness of functional languages such as ML [58] or Haskell [36] with correct-by-construction session-typed primitives that guarantee deadlock-freedom and execution progress.

By taking advantage of the logical foundation of session types, we can adopt a modular approach to our language design, starting with a pure functional and concurrent language core, and extending it with more expressive features such as value-dependent or refined session types [4, 56], asynchronous channels [7], and even novel features such as unsafe (or gradually-checked) code points.

Contributions The main contributions of this work will be:

- A type checker for our concurrent language, featuring state-of-the-art bidirectional type checking techniques [10];
- A concurrent interpreter for our language;
- A compiler for our language, targeting a real-world "backend" (e.g. JVM, LLVM), and a runtime to manage the session type logic of our language.

1.3 Document Structure

The rest of this document is structured as follows. Chapter 2 contains the theoretical background for the proposed work - it briefly presents concurrent programming, and describes different process calculi, introducing several typed variants. Chapter 3 details different existing practical implementations of session types, along with their respective limitations in ensuring statical safety guarantees. Chapter 4 contains our proposal of a language that implements logical session types, outlines the details of different parts of the proposal, and defines a concrete timetable specifying the execution of our proposal.

Chapter 2

Background

2.1 Concurrent Programming

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:

- 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, where the program's various processes execute 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, since the possible interactions between these processes open new possible execution paths to incorrect program states. As such, a major difficulty 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 [1]. 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.

2.2 Process Calculi

Process calculi are formal languages used to model and reason about concurrent (message-passing) programs by abstracting away all but the concurrent aspects of a program. These concurrent features are modeled through the use of processes that exchange messages among themselves; in fact, the communication between processes represents the essential, indivisible, component in process calculi. It's important to note that although these calculi are extremely competent at faithfully modeling concurrent interaction of processes in a abstract, mathematical, manner, rarely do they map straightforwardly to a real-world concurrent programming language. As an introduction to process calculi, two examples are presented below: first the Calculus of Communicating Systems (CCS) [38], and then the π-calculus [51], both of which introduce the notion of *communication channel*, an abstraction of a communication link between two processes. CCS was one of the first process calculi to be developed and, as a result, is relatively simple when compared to its other successor calculi. The π-calculus appeared later, offering the ability for communication channels to be sent through channels themselves and paving the way for the modeling of higher-level process interactions. The π-calculus is also Turing-complete, which means it is more expressive than CCS, which is not. Thus, the π-calculus came to be the *de facto* process calculus to use in concurrent program modeling.

2.2.1 Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) [38] is a basic process calculus developed for the purpose of describing concurrent systems. CCS was built according to two core ideas:

- First, that of *observation*. It should be possible to describe the behavior of a process (called an agent in CCS) as it is experienced by an external observer, regardless of its inner, unseen, workings. If two agents have identical behavior (i.e. evolution

$$\alpha ::= \tau \mid a \mid \bar{a}$$

$$B ::= nil \mid B + B' \mid \alpha B \mid B|B' \mid B \setminus \alpha \mid B[a_1/a_2] \mid id$$

Figure 21: CCS Syntax

in state) as perceived by an observer, those agents are said to be *observationally equivalent*;

- Second, that of *synchronized communication*. The communication between independent agents is regarded as the essential building block in concurrent systems, so CCS offers a binary operation, called concurrent composition, which composes two independent agents and allows them to communicate via common channels.

The syntax of CCS is presented in Figure 21. CCS consists of the following behavior expressions to describe an agent's behavior: an agent that does nothing, written nil; the sum of two behaviors, written $B + B'$, in which an agent proceeds exclusively as B or B'; performing of an action, written αB, in which an agent performs action α and continues as B; the composition of two behaviors, written $B|B'$, in which two agents, B and B', run in parallel; the restriction of behavior, written $B \setminus \alpha$, in which an agent proceeds as B without action α; the relabeling of actions, written $B[a_1/a_2]$, in which an agent proceeds as B with actions a_2 renamed to a_1; variables representing an agent, written id, an agent proceeds as the agent identified by id. As for actions, α, in CCS, there are dual actions, for instance a and $\bar{a}$, which allow two agents to synchronize when one of the agents performs one of the dual actions, and the other agent performs the other of the dual actions; there is also the action τ, which represents an internal action of an agent. which can not be perceived by external observers.

2.2.2 π-Calculus

The π-calculus [31, 32, 51] is a process calculus used to model concurrency in programs through the use of processes that exchange messages among themselves. Unlike CCS, where processes can only synchronize over names, in the π-calculus channels carry *payloads*. The π-calculus defines two types of payloads: *names* - which represent communication channels - and *processes*. Processes perform actions and evolve to different processes. Processes use names or channels to interact with each other and may pass names from one to to another during these interactions.

The π-calculus is notable for its minimal, yet expressive syntax as seen in Figure 22. Processes behavior can be described in the following ways: the process that can do nothing, written 0; the performing of an action, written $\pi.P$, in which a process performs action π and proceeds as process P ; the sum of behaviors, written $P + P'$, in which a process proceeds exclusively as P or P', the parallel composition, written $P|P'$, in which the processes P and P' run in parallel; scope restriction, written $(\nu z)P$, in which the scope

$$\pi ::= \bar{x}y \mid x(z) \mid \tau \mid [x = y]$$

$$P ::= 0 \mid \pi.P \mid P + P' \mid P|P' \mid (\nu z)P \mid !P$$

Figure 22: π-calculus Syntax

of z is restricted to process P; replication, written $!P$, representing an infinite number of parallel copies of P. In the π-calculus the actions - π - performed by processes can be of the following forms: send name y via channel x, written $\bar{x}y$; receive name z from channel x, written $x(z)$; unobservable action, written τ; conditionally check if x and y are the same, written $[x = y]$.

2.3 Typed Languages

A type system is a mechanism used to classify the expressions of a language. Type systems for programming languages, be they sequential or concurrent, are useful for several reasons:

- to detect programming errors statically - e.g. invalid type assignments, incorrect parameter type passing;
- to extract information useful for reasoning about the behavior of programs - if the type of a value is known, its possible to infer what operations one can apply over that value;
- to improve the efficiency of compiled code - if the type of a value (e.g. int) is known at compile-time, the exact amount of memory is statically allocated for it, without any waste of space;
- for documentation purposes - types are a form of documentation of the intent of the programmer when writing code.

Given the noted benefits of type systems, their practical implementation was of great interest; the first step in accomplishing this goal is defining a theoretical basis for type systems and establishing their soundness properties. An attractive way to do this in a concurrent setting is by extending a process calculus with a type system. Since the π-calculus is regarded as the process calculus of choice, it was the prime candidate for typing - indeed, several variants of typed π-calculus were developed, many of which are presented below.

2.3.1 Typed π-Calculus

Type systems rely on type judgments, asserting that language expressions are well-typed. In the π-calculus, types are assigned to communication channels. Type judgments are

often written $\Gamma \vdash P : T$, where Γ is a type environment and P may either be a process or a value. The type environment Γ is a finite set of type assignments, where the names in the assignments are all distinct. A type assignment is written $P : T$, where P is a name of a process or a value and T is a type. To refer to the type assigned to P by Γ, we write $\Gamma(P)$. If P is a process, then T is $\diamond$, the behavior type. A process type judgment $\Gamma \vdash P : \diamond$ asserts that process P respects all the type assumptions in Γ. A value type judgment $\Gamma \vdash v : T$ asserts that value v has type T under the type assumptions Γ. For a given type system, a valid type judgment is one that can be proved from the axioms and inference rules of the type system - a proof of the validity of a type judgment is a typing derivation. In the π-calculus, a process or value P is said to be well-typed in Γ if there is a T such that $\Gamma \vdash P : T$ is valid.

We now describe several type systems for the π-calculus: Base-π, the most basic of typed π-calculus variants, offering only typed channels on which to send primitive values; the simply-typed π-calculus, an evolution over Base-π, capable of typing channel-passing; I/O types, which can statically enforce directionality of communication; types for deadlock-freedom and respective refinements; session types, which model reciprocal interactions between participants; and logical session types, which result from an equivalence with intuitionistic linear logic and augment session types with stronger correctness properties.

2.3.1.1 Base-π

As a first approach to a typed π-calculus, we introduce Base-π, essentially the core of CCS plus value-passing. In Base-π, channel payloads are typed, meaning channels transmit data of a certain primitive type, and only that type (e.g. integers, booleans, or tuples). The syntax of Base-π is presented in Figures 23 and 24, which define the syntactic rules for types and processes in Base-π, respectively.

Base-π enforces that well-typed processes remain well-typed as they evolve during communication with each other and link *arity* (number of arguments sent in the channel) is strictly enforced, meaning processes can not misuse channels by sending values of unexpected types.

2.3.1.2 Simply-Typed π-Calculus

The simply-typed π-calculus is an extension of Base-π where value types are enriched to allow for channels to be communicated from process to process. This is achieved by considering type L as value types. For instance, $\#(\#B)$ is the type of a channel that can transmit channels of type $\#B$, which are channels that can transmit basic values of type B.

			Types
S, T	::=	V	value type
	|	L	link/channel type
	|	$\diamond$	behavior type
			Value Types
V	::=	B	basic type
			Channel Types
L	::=	$\#V$	connection type
			Type Environments
Γ	::=	$\Gamma, x : L$	type environment union
	|	$\Gamma, x : V$	type environment union
	|	$\emptyset$	empty type environment

Figure 23: Base-π Type Syntax

			Values
v, w	::=	x	name
	|	val	basic value
			Prefixes
π	::=	$v(x)$	input
	|	$\bar{v}w$	output
	|	τ	silent prefix
	|	$[v = w]\pi$	matching
			Processes
P, Q, R	::=	$(\nu x : L)P$	restriction
	|	M	summation
	|	$P \mid P'$	composition
	|	$!P$	replication
			Summations
M	::=	0	process that can do nothing
	|	πP	do π and proceed as P
	|	$M + M'$	proceed exclusively as M or M'

Figure 24: Base-π Process Syntax

$$\text{(Input)}\quad \frac{\Gamma \vdash a : iT \qquad \Gamma, x : T \vdash P : \diamond}{\Gamma \vdash a(x).P : \diamond} \qquad \text{(Output)}\quad \frac{\Gamma \vdash a : oT \qquad \Gamma \vdash w : T \qquad \Gamma \vdash P : \diamond}{\Gamma \vdash \bar{a}w.P : \diamond}$$

Figure 25: Typing Rules for I/O Types

2.3.1.3 I/O Types

The π-calculus can be further refined by enriching it with so called *I/O types* [51], meaning that channels are specifically typed as being used for input, output or both. I/O types consist of the following type constructors: oT, the type of a channel that can be used only for output and carries values of type T; iT, the type of a channel that can be used only for input and carries values of type T, and $\#T$, the type of a channel that is used for both input and output and carries values of type T. The additional typing rules for I/O types appear in Figure 25. The input rule states that if a process that receives x on channel a and proceeds as P under the assumptions of Γ is well-typed, then process P is well-typed in $\Gamma, x : T$, and channel a has type iT under Γ. The output rule states that if a process that sends w on channel a and continues as P is well-typed under the assumptions of Γ, then process P is well-typed, channel a is of type oT, and value w is of type T, all under the assumptions of Γ.

I/O types are important because they make it possible to enforce *polarity* constraints or invariants on names, preventing processes from misusing channels by sending values through a channel that should be used only for receiving or vice-versa. As an example of a practical implementation, the Go programming language [8] uses similar mechanisms to I/O types in its typed concurrent communication primitives.

2.3.1.4 Linear Types

Linear type systems [51] are designed to provide fine-grained control of resource usage by limiting the number of times a given resource is used (in the case of π-calculus, the resources are channels). Linear types in the π-calculus [51] can be seen as a further refinement of I/O types, allowing for the imposition of both *polarity* constraints - what kind of operation (input, output, both) can a channel be used for - and *multiplicity* constraints - how many times a channel can be used. There are three kinds of type constructors in linear types: $\ell_o T$, which gives a channel the capability of being used once for output; $\ell_i T$, which gives a channel the capability of being used exactly once for input; and $\ell_\# T$, which gives a channel the capability of being used once for output and once for input. One limitation in linear typing should be noted: though a well-typed process P will never use a linear channel c more than once, it may happen that P never reaches a state where it uses c, meaning there is no guarantee that c will be used once, and only once, for sure. Which means that under a linear type system, there is only the following guarantee: *if* a channel is used, it's used *exactly once*.

Something important to take note of in linear type systems is that, in the case of parallel composition, for instance of $P_1 \mid P_2$ under a typing Γ, it must be enforced that linear capabilities in Γ are exercised only once, either in P_1 or in P_2. The way to do this is to split the linear capabilities in Γ and derive two new type environments, Γ_1 and Γ_2, under which P_1 and P_2 are typed, respectively, such that Γ is a combination of Γ_1 and Γ_2, as seen below:

$$\text{(LIN-PAR)} \quad \frac{\Gamma_1 \vdash P_1 : \diamond \qquad \Gamma_2 \vdash P_2 : \diamond}{\Gamma_1 \uplus \Gamma_2 \vdash P_1 \mid P_2 : \diamond}$$

The *combination* operation ($\uplus$) works in the following way for linear types:

$$\begin{array}{lcl} \ell_i T \uplus \ell_o T & \stackrel{def}{=} & \ell_\# T \\ T \uplus T & \stackrel{def}{=} & T \end{array}$$

For instance, take two types $\Gamma_1(x)$ and $\Gamma_2(x)$, which are the types assigned to x by Γ_1 and Γ_2, respectively. If $\Gamma_1(x)$ and $\Gamma_2(x)$ are linear types of opposite polarities, they combine into a linear type; if $\Gamma_1(x)$ and $\Gamma_2(x)$ are the same non-linear type, they combine into a non-linear type $\Gamma_1(x) \uplus \Gamma_2(x)$; and if $\Gamma_1(x)$ and $\Gamma_2(x)$ are the same linear type, their combination is undefined, that is, there is no way to split a type environment Γ and obtain such combination.

2.3.1.5 Types for Deadlock-Freedom

In a concurrent setting, it is possible for a process P to reach a state in which it attempts to perform an input or output operation for which there is no matching output/input action in a parallel process. In this situation, P blocks, waiting for communication, unable to evolve to its next state. If it is the case that all existing processes are in these same circumstances, waiting to perform communication with one another, none of them managing to progress, then they are said to be in a *deadlock*.

Linearly-typed π-calculus offers no guarantees about the absence of deadlocks in its programs. Take, for instance the process $(\nu z : \ell_\# T) z(x).\bar{c}x$, where c is a linear channel. The process creates a new channel z and forwards a message x from z through c afterwards; since z is a new channel, private to this process, there will be no other process sending through z, which means the process will be left waiting forever on z, never reaching the point where c will be used - the process is deadlocked.

Absence of deadlocks - or deadlock-freedom - is a fundamental correction property of programs. Furthermore, it is a property that can not be enforced dynamically, since at that stage of execution, the process configuration is already stuck. Thus there is great appeal in the development of static techniques, such as type systems, that can ensure deadlock-freedom before a program runs. As such, more type systems were developed, further refining π-calculus. Most of these work by enforcing a temporal partial ordering on process communication, through the annotation of channel type constructors with

time tags, like so: $x : \ell_{\#} T,\ y : \ell_{\circ} T,\ \{(t_x, t_y)\} \vdash x(z).\bar{y}z$. The tag $\{(t_x, t_y)\}$ means that "communication on y may be delayed until communication on x is finished". If the time tags of two processes can be consistently ordered, then the processes are deadlock-free, otherwise there is a cyclic ordering of the time tags, and the processes deadlock. For example, in the following process, which is a parallel composition of two processes, $x(z).\bar{y}z \mid y(z).\bar{x}z$, there is a cyclical ordering of the time tags $\{(t_x, t_y), (t_y, t_x)\}$, therefore the process will deadlock.

The partial ordering of communication is not a perfect strategy for deadlock detection [26]. For instance, the process $(\nu x : \ell_{\#} T)x(z).0$ has a partial ordering of time tags, since there is only one channel, but it will never progress; in this case the deadlock was not detected. Another example is the process $P = x(z).(\bar{y}a \mid \bar{z}b) \mid y(c).\bar{x}w$, which presents a cyclic dependency of time tags; however if there is some other process $\bar{x}w \mid w(d).x(z).0$ in parallel with P, then P will not deadlock - in this case a deadlock was incorrectly detected. It is therefore necessary to employ additional techniques to complement partial ordering in deadlock detection. An example of one such technique is the classification of channels according to their usage, and forced enforcement of said usage by the type system. A concrete, instance of this technique is to classify channel usage into three types of channels - called *reliable channels*: *linear channel* - used once for sending and once for receiving, *replicated input channel* - may only appear once in a replicated process when used to receive, but may be used any number of times to send, and *mutex channel* - functions similarly to a binary semaphore, a message must be sent after it is created and a process that has received from it must eventually send through it. The combination of partial ordering along with the exclusive use of reliable channels ensures deadlock-freedom (if unreliable channels are used besides the reliable channels, then deadlock-freedom can not be guaranteed).

Type systems for deadlock-freedom can be further refined by additionally classifying communication channels according to their *capability*, that is what can they be used for (either sending, receiving or both), and their *obligation*, i.e. what they must absolutely be used for. This classification of channels allows for the capture of increased inter-channel dependency information, thus making it possible for the typification of more complex concurrent interactions.

2.3.1.6 Refining Deadlock-Freedom

A process *lock* is subtly different from a regular deadlock. As an example, take the process P, which is a parallel composition of three other processes P_1, P_2, and $c(x).P_3$. Suppose that P_1 and P_2 are continuously communicating with each other, repeating the same communication steps in a loop. In that case, $c(x).P_3$ never receives anything on channel c, and so it can not progress. P is not deadlocked, since there is ongoing communication occurring between P_1 and P_2, however, P is definitely *locked*, since no actual progress is being made: P_1 and P_2 communicate but do not progress towards any goal, and $c(x).P_3$ does not progress at all.

Type systems for lock-freedom are typically built on top of type systems for deadlock-freedom, and enriched with additional mechanisms. In one such type system for lock-freedom [27], channel types are augmented with the information about how much "time" it takes for a process to become ready to input or output a value on a channel and how much "time" it takes for the process to succeed in its I/O operation after it becomes ready. This type system enforces lock-freedom for certain types of communications, however it does so only under the assumption that the scheduling scheme for running programs is *strongly fair* (meaning that every process that is able to participate in a communication infinitely often can eventually participate in the communication), which is not always realistic. Another possible type system [30] combines the functionality of type systems for deadlock-freedom with *termination* mechanisms, to ensure termination of processes. This hybrid approach turns out to be quite complex and hard to scale to the enforcement of type correction in very large programs.

As we have argued above, type systems for lock-freedom [27, 30] are typically more complex than type systems for deadlock-freedom - due to the fact that it's harder to enforce liveness than it is to enforce safety - and sometimes have narrow applications, or can only work under stricter assumptions, which limits their usefulness. Moreover, it's difficult to see, from a practical point of view, how to map these complex type systems to actual general-purpose languages - for instance, in the case of the first mentioned type system for lock-freedom [27] there isn't any way for types to be inferred, since they rely on "manual" temporal bounding; furthermore it's hard to reason how should the implementation of such types should actually be done.

2.3.2 Session Types

Among the communication primitives in the previously presented systems, there are no type-level constructs that allow us to to structure a series of reciprocal interactions between two parties. Meaning, the only way to model a series of communications following a certain scenario (e.g. the interactions between a file server and its client) is to describe them as distinct, unrelated interactions. In applications which have complex reciprocal interactions between concurrent processes, the lack of structuring methods might lead to errors, especially protocol compliance errors. Session type systems were 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 [17]. 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

$$
\begin{array}{rcl}
P & ::= & request\ a(x)\ in\ P \mid accept\ a(x)\ in\ P \mid k![\tilde{e}];P \mid k?(\tilde{x})\ in\ P \\
& \mid & k \triangleleft l;P \mid k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\} \mid throw\ k[k'];P \mid catch\ k(k')\ in\ P \\
& \mid & if\ e\ then\ P\ else\ Q \mid P \mid Q \mid inact \mid (\nu u)P \mid X[\tilde{e}\tilde{k}] \mid def\ D\ in\ P \\
\\
D & ::= & X_1(\tilde{x}_1\tilde{k}_1) = P_1\ and\ \ldots\ and\ X_1(\tilde{x}_n\tilde{k}_n) = P_n
\end{array}
$$

Figure 26: Process Language Syntax for Session π-Calculus [17]

when initiating each session.

The syntax of an early session process calculus [17] is presented in Figure 26. Requesting the start of a session is written $request\ a(k)\ in\ P$ - the request for the start of a new session is sent, via the name a, and a new channel k is generated to be then used in later communications in the process P. Reception of a session request is written $accept\ a(k)\ in\ P$ - a request for session initiation is received via a and a fresh channel k is generated, to be used for further communications in P. The sending of data is written $k![\tilde{e}];P$ - data $\tilde{e}$ (may be a tuple) is sent through the channel k and the process continues as P. Reception of data is written $k?(\tilde{x})\ in\ P$ - data is received through channel k and the process continues as P with $\tilde{x}$ now bound to the received data. Label selection is written $k \triangleleft l;P$ - a label is selected through channel k and execution continues as process P. Branching is written $k \triangleright \{l_1 : P_1 \mid ... \mid l_n : P_n\}$ - various labels, each matching a different process, are offered on channel k. The sending of a channel is written $throw\ k[k'];P$ - channel k' is sent through channel k, and execution proceeds as process P. Reception of a channel is written $catch\ k(k')\ in\ P$ - a channel is received through k, and continuation as process P with k' bound to the received channel. The conditional construct is written $if\ e\ then\ P\ else\ Q$ - execution proceeds as P or Q depending on the value of e. Parallel composition is written $P \mid Q$ - execute processes P and Q in parallel. Process inaction is written $inact$. Name/channel hiding is written $(\nu u)P$ - u is exclusive to process P. Process variables are written $X[\tilde{e}\tilde{k}]$. Definition of recursion is written $def\ D\ in\ P$.

The types of this session calculus [17] are shown in Figure 27, with sorts S, types α, β, sort variables s, and type variables t. A sort $\langle \alpha, \bar{\alpha} \rangle$ represents two complementary, or dual, structures of interaction associated with a channel. The type $\downarrow[\tilde{S}];\alpha$ represents the type of a channel that receives a channel with a sort $\tilde{S}$ and continues as type α. Type $\downarrow[\alpha];\beta$ represents the type of a channel used to receive a value of type α and proceed as type β. Type $\uparrow[\tilde{S}];\alpha$ represents the type of a channel that is used to send a channel with sort $\tilde{S}$ and proceeds as type α. Type $\uparrow[\alpha];\beta$ represents the type of a channel used to send a value of type α and proceed as type β. Type $\&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}$ represents the type of a session that offers a choice between the behaviors $\alpha_1, \ldots, \alpha_n$, identified by the labels $l_1, \ldots, l_n$, respectively. Type $\oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}$ is the type of a session that provides a label in $l_1, \ldots, l_n$, which identify behaviors $\alpha_1, \ldots, \alpha_n$, respectively. Type $\mathbf{1}$ is the type of channel that has terminated. Type $\perp$ is the type of a channel that allows no further connection.

$$
\begin{array}{rcl}
S & ::= & \text{nat} \mid \text{bool} \mid \langle \alpha, \bar{\alpha} \rangle \mid s \mid \mu s.S \\
\\
\alpha & ::= & \downarrow[\tilde{S}];\alpha \mid \downarrow[\alpha];\beta \mid \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid 1 \mid \bot \\
& \mid & \uparrow[\tilde{S}];\alpha \mid \uparrow[\alpha];\beta \mid \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid t \mid \mu t.\alpha
\end{array}
$$

Figure 27: Type Syntax for Session π-Calculus [17]

The described session type system 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.

It should be noted that, while session type systems assure deadlock-freedom, they only do so in the case that two processes only maintain 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 (e.g. a scenario between two buyers and one seller, all communicating with each other in sequence; the notion of sequence of communication is abstracted as three separate binary sessions, instead of a single session). To address this problem, *multiparty* session types [18] were proposed.

Multiparty session types abstract interactions between several participants, in an asynchronous setting. The process behavior for a multiparty type system is present in Figure 28. Process behavior is similar to the previously presented dyadic session type system, but there is some behavior of note: multicast session request, written $\bar{a}[2..n](\tilde{s}).P$ - distribute fresh session channels among all $2..n$ participants, and proceed as P; session delegation, written $s!\langle\langle\tilde{s}\rangle\rangle;P$ - delegate the capability to participate in sessions, by passing channels $\tilde{s}$ associated with the sessions, and continue as P; session reception, written $s?((\tilde{s}));P$ - receive existing channel sessions and continue as P; and message queuing written $s :: \tilde{h}$ - ordered messages in transit $\tilde{h}$ with destination s.

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. However, there are disadvantages to a multiparty session type system: assuring that a *global* type is well-formed can rule out many morally correct protocols; it's not compositional - the number of session participants is fixed and they must

$$
\begin{array}{rcll}
P & ::= & \tilde{a}[2..n](\tilde{s}).P & \text{multicast session request} \\
 & | & a[p](\tilde{s}).P & \text{session acceptance} \\
 & | & s!\langle \tilde{e} \rangle; P & \text{value sending} \\
 & | & s?(\tilde{x}); P & \text{value reception} \\
 & | & s!\langle\langle \tilde{s} \rangle\rangle; P & \text{session delegation} \\
 & | & s?((\tilde{s})); P & \text{session reception} \\
 & | & s \triangleleft l; P & \text{label selection} \\
 & | & s \triangleright \{l_i : P_i\}_{i \in I} & \text{label branching} \\
 & | & \text{if } e \text{ then } P \text{ else } Q & \text{conditional branch} \\
 & | & P \mid Q & \text{parallel composition} \\
 & | & 0 & \text{inaction} \\
 & | & (\nu n)P & \text{hiding} \\
 & | & \mathit{def}\ D\ \mathit{in}\ P & \text{recursion} \\
 & | & X\langle \tilde{e}\tilde{s} \rangle & \text{process call} \\
 & | & s :: \tilde{h} & \text{message queue} \\
D & ::= & \{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i \in I} & \text{declaration for recursion}
\end{array}
$$

Figure 28: Process Syntax for Multiparty Session Type System [18]

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.

2.3.3 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 [51], types for deadlock-freedom [26] and its refined variations [27, 30], and session types [17, 18], 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, 46], 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. The set of processes which make up the typed calculus is defined by

$$P ::= P \mid Q \mid (\nu y)P \mid x\langle y \rangle.P \mid x(y).P \mid !x(y).P \mid x.inl; P \mid x.inr; P \mid x.case(P, Q)$$

The operators 0, $P \mid Q$, and $(\nu y)P$ are the same as those in π-calculus. As for the remaining operators: $x\langle y \rangle.P$ represents the sending of name y on channel x, and continued execution as P; $x(y).P$, which means a process receives a name on x and binds it to y, then proceeds as

P; $!x(y).P$, the replicated input construct; $x.case(P, Q)$ represents the offering of branching choice on channel x; $x.inl; P$ represents the choice of the "left" alternative provided by x and continuation as P; $x.inr; P$ represents the choice of the "right" alternative provided by x and continuation as P. In this type system, types are assigned to names, which represent communication channels, and describe their communication protocol. The types that exist in this type system are the syntax of linear logic properties and are given by

$$A, B ::= \mathbf{1} \mid !A \mid A \otimes B \mid A \multimap B \mid A \oplus B \mid A \& B$$

The type $\mathbf{1}$ represents a terminated session. The type $!A$ represents a channel to be used by a server for creating an unlimited, but finite, number of sessions of type A. $A \otimes B$ represents the type of a session that performs an output of type A and then proceeds as B. $A \multimap B$ is the type of a session that first receives an input of type A and then proceeds as B. $A \oplus B$ is the type of a session that either chooses to proceed as A ("left" choice) or as B ("right" choice). $A \& B$ is the type of a session that offers a choice between behaviors of type A ("left" choice) or B ("right" choice).

This type system distinguishes between two kinds of type environments: a *linear* type environment Δ and an *unrestricted* type environment Γ, where weakening and contraction principles hold for Γ but not for Δ. 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. We refer to $\Gamma; \Delta$ and $z : C$ as the left-hand and right-hand side typings, respectively. 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 of deadlocks and communication errors. The rules of the type system are presented in Figure 29. T and S represent right-hand side singleton environments (e.g. $z : C$).

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. Indeed, take the the sequent calculus shown in Figure 210. Sequents have the form $\Gamma; \Delta \vdash D : C$, where Γ is the unrestricted context, Δ the linear context, C a 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 $\Gamma; \Delta \vdash D : A$. This type system ensures *session fidelity* - processes send and receive data correctly, according to the type of the session channel - and provides *global progress* 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.

$$(\text{T1L})\frac{\Gamma;\Delta \vdash P :: T}{\Gamma;\Delta, x : 1 \vdash P :: T} \qquad (\text{T1R})\frac{}{\Gamma;\cdot \vdash 0 :: x : 1}$$

$$(\text{T}\otimes\text{L})\frac{\Gamma;\Delta, y : A, x : B \vdash P :: T}{\Gamma;\Delta, x : A \otimes B \vdash x(y).P :: T} \qquad (\text{T}\otimes\text{R})\frac{\Gamma;\Delta \vdash P :: y : A \qquad \Gamma;\Delta' \vdash Q :: x : B}{\Gamma;\Delta,\Delta' \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: x : A \otimes B}$$

$$(\text{T}\multimap\text{L})\frac{\Gamma;\Delta \vdash P :: y : A \qquad \Gamma;\Delta', x : B \vdash Q :: T}{\Gamma;\Delta,\Delta', x : A \multimap B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: T} \qquad (\text{T}\multimap\text{R})\frac{\Gamma;\Delta, y : A \vdash P :: x : B}{\Gamma;\Delta \vdash x(y).P :: x : A \multimap B}$$

$$(\text{Tcut})\frac{\Gamma;\Delta \vdash P :: x : A \qquad \Gamma;\Delta', x : A \vdash Q :: T}{\Gamma;\Delta,\Delta' \vdash (\nu x)(P \mid Q) :: T} \qquad (\text{Tcut}^!)\frac{\Gamma;\cdot \vdash P :: y : A \qquad \Gamma, u : A;\Delta \vdash Q :: T}{\Gamma;\Delta \vdash (\nu u)(!u(y).P \mid Q) :: T}$$

$$(\text{T!L})\frac{\Gamma, u : A;\Delta \vdash P\{u/x\} :: T}{\Gamma;\Delta, x :!A \vdash P :: T} \qquad (\text{T!R})\frac{\Gamma;\cdot \vdash Q :: y : A}{\Gamma;\cdot \vdash !x(y).Q :: x :!A}$$

$$(\text{T}\oplus\text{L})\frac{\Gamma;\Delta, x : A \vdash P :: T \qquad \Gamma;\Delta, x : B \vdash Q :: T}{\Gamma;\Delta, x : A \oplus B \vdash x.case(P, Q) :: T} \qquad (\text{T}\&\text{L}_2)\frac{\Gamma;\Delta, x : B \vdash P :: T}{\Gamma;\Delta, x : A\&B \vdash x.inr; P :: T}$$

$$(\text{T}\&\text{R})\frac{\Gamma;\Delta \vdash P :: x : A \qquad \Gamma;\Delta \vdash Q :: x : B}{\Gamma;\Delta \vdash x.case(P, Q) :: x : A\&B} \qquad (\text{T}\&\text{L}_1)\frac{\Gamma;\Delta, x : A \vdash P :: T}{\Gamma;\Delta, x : A\&B \vdash x.inl; P :: T}$$

$$(\text{T}\oplus\text{R}_1)\frac{\Gamma;\Delta \vdash P :: x : A}{\Gamma;\Delta \vdash x.inl; P :: x : A \oplus B} \qquad (\text{T}\oplus\text{R}_2)\frac{\Gamma;\Delta \vdash P :: x : B}{\Gamma;\Delta \vdash x.inr; P :: x : A \oplus B}$$

$$(\text{Tcopy})\frac{\Gamma, u : A;\Delta, y : A \vdash P :: T}{\Gamma, u : A;\Delta \vdash (\nu y)u\langle y\rangle.P :: T}$$

Figure 29: Typing Rules for Logical Session Types

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 [46], such as termination, behavioral equivalence, confluence and parametricity; moreover, logical session type systems are compositional.

2.3.3.1 Beyond the Correspondence Between Intuitionistic Linear Logic and Session Types

One fundamental advantage of the correspondence between intuitionistic linear logic and session types is its flexibility; that is there are different ways to approach this correspondence that yield possibly more expressive type systems. As examples of such different approaches and type systems, we highlight the *asynchronous interpretation* of the correspondence between linear logic and session types [7], and dependent session types [55,

$$(1\mathrm{L})\frac{\Gamma;\Delta \vdash D : C}{\Gamma;\Delta, x : 1 \vdash 1L\ x\ D : C} \qquad (1\mathrm{R})\frac{}{\Gamma;\cdot \vdash 1R : 1}$$

$$(\otimes\mathrm{L})\frac{\Gamma;\Delta, y : A, x : B \vdash D : C}{\Gamma;\Delta, x : A \otimes B \vdash \otimes L\ x\ (y.x.\ D) : C} \qquad (\otimes\mathrm{R})\frac{\Gamma;\Delta \vdash D : A \qquad \Gamma;\Delta' \vdash E : B}{\Gamma;\Delta,\Delta' \vdash \otimes R\ D\ E : A \otimes B}$$

$$(\multimap\mathrm{L})\frac{\Gamma;\Delta \vdash D : A \qquad \Gamma;\Delta', x : B \vdash E : C}{\Gamma;\Delta,\Delta', x : A \multimap B \vdash \multimap L\ x\ D(x.\ E) : C} \qquad (\multimap\mathrm{R})\frac{\Gamma;\Delta, y : A \vdash D : B}{\Gamma;\Delta \vdash \multimap R\ (y.\ D) : A \multimap B}$$

$$(\mathrm{cut})\frac{\Gamma;\Delta \vdash D : A \qquad \Gamma;\Delta', x : A \vdash E : C}{\Gamma;\Delta,\Delta' \vdash cut\ D\ (x.\ E) : C} \qquad (\mathrm{cut}^!)\frac{\Gamma;\cdot \vdash D : A \qquad \Gamma, u : A;\Delta \vdash E : C}{\Gamma;\Delta \vdash cut^!\ D\ (u.\ E) : C}$$

$$(!\mathrm{L})\frac{\Gamma, u : A;\Delta \vdash D : C}{\Gamma;\Delta, x :!A \vdash !L\ x\ (u.\ D) : C} \qquad (!\mathrm{R})\frac{\Gamma;\cdot \vdash D : A}{\Gamma;\cdot \vdash !R\ D :!A}$$

$$(\oplus\mathrm{L})\frac{\Gamma;\Delta, x : A \vdash D : C \qquad \Gamma;\Delta, x : B \vdash E : C}{\Gamma;\Delta, x : A \oplus B \vdash \oplus L\ x\ (x.\ D)(x.\ E) : C} \qquad (\&\mathrm{L}_2)\frac{\Gamma;\Delta, x : B \vdash D : C}{\Gamma;\Delta, x : A\&B \vdash \&L_2\ x\ (x.\ D) : C}$$

$$(\&\mathrm{R})\frac{\Gamma;\Delta \vdash D : A \qquad \Gamma;\Delta \vdash E : B}{\Gamma;\Delta \vdash \&R\ D\ E : A\&B} \qquad (\&\mathrm{L}_1)\frac{\Gamma;\Delta, x : A \vdash D : C}{\Gamma;\Delta, x : A\&B \vdash \&L_1\ x\ (x.D) : C}$$

$$(\oplus\mathrm{R}_1)\frac{\Gamma;\Delta \vdash D : A}{\Gamma;\Delta \vdash \oplus R_1\ D : A \oplus B} \qquad (\oplus\mathrm{R}_2)\frac{\Gamma;\Delta \vdash D : B}{\Gamma;\Delta \vdash \oplus R_2\ D : A \oplus B}$$

$$(\mathrm{copy})\frac{\Gamma, u : A;\Delta, y : A \vdash D : C}{\Gamma, u : A;\Delta \vdash copy\ u\ (y.\ D) : C}$$

Figure 210: Dual Intuitionistic Linear Logic

56].

Asynchronous Interpretation It has been shown that intuitionistic linear logic can be regarded as a session-type discipline for the π-calculus, where cut reduction corresponds to synchronous process reductions [14, 16]. Later work [7] has proved that the correspondence of intuitionistic linear logic to a session-type discipline also holds in an asynchronous setting, where cut reductions correspond to a natural asynchronous buffered session semantics, where each session is allocated a separate communication buffer. This result is interesting since asynchronous communication can be seen as a more realistic model for concurrency, especially since realistic implementations of communication channels generally use buffers, and thus being able to establish properties of asynchronous processes by static typing is of great importance. Furthermore, this asynchronous interpretation exposes additional parallelism inherent in linear logic that remained absent in

the synchronous interpretation.

Dependent Session Types Dependent session types [55, 56] are an extension of the correspondence between intuitionistic linear logic and session types to first and higher-order linear logic. Dependent session types allow for the expression of rich constraints on sessions which go beyond regular session type systems. It's possible to specify not only the dynamics of protocols, but also the properties of data received in sent in communications. As an example, take a *Server* that takes an integer and returns its square; under the original session type system, the type of *Server* would be

$$int \multimap int \otimes \mathbf{1}$$

which, simply means that the *Server* receives an integer and returns an integer. However, with dependent session times, *Server* would have be typed like so

$$\forall x : int.\ \exists y : int.\ y = x^2.\mathbf{1}$$

which means that the *Server* receives an integer and returns its square. Moreover, with dependent session types, it's possible to implement a scheme of proof-carrying certification in the communication between two untrusted parties, and additionally implement a method of proof irrelevance to identify proofs that may be safely erased at runtime (in the case that the communicating parties are trusted), decreasing communication overhead imposed by the system.

Chapter 3

Related Work

In this chapter, we report on several existing implementations of session types. These can be broadly divided into two categories: those that are implemented on top of existing general-purpose languages; and those that are implemented on top of a completely new language. We present both kinds of implementations. The common theme across all of the presented implementations is their inability to provide strong statical guarantees about program safety and liveness.

3.1 Implementation of Session Types in Real-World Languages

One major obstacle to the adoption and integration of session type systems into mainstream programming languages is their reliance on sophisticated and peculiar features, whose built-in support would require massive changes in both languages and their development tools [44]. Thus, several efforts were made to implement session types in general-purpose-languages in practical ways: as libraries, with specific APIs, which provide certain guarantees given that the APIs are used correctly; as libraries with additional DSL specification and code generation; and by leveraging advanced features of the host language type system.

3.1.1 Session Types as Libraries

Session types are often implemented in general-purpose languages as libraries, with specific APIs. These libraries can then be used in programs of the target languages, and provide guarantees of certain properties (such as communication safety, protocol fidelity, deadlock-freedom and progress), provided that the APIs are used correctly. Note, however, that few, if any of these properties can be guaranteed statically: correct API usage needs to be checked at runtime, and some properties (e.g. linearity) can not be fully

ensured. There are several examples of implementations of session types as libraries in various real-world languages, such as in OCaml [44], and Haskell [34, 42]. Library implementations of session types tend to be lightweight, since there is no need for additional user effort besides importing the library features into a program. This kind of implementations tend to be more general-purpose, and therefore less tailored to specific protocols that a user may wish to program. The main disadvantage of library implementations is the general lack of static correctness guarantees.

3.1.2 Session Types as Libraries with Code Generation

Session types do not have to be implemented solely as standalone libraries. Their implementation can consist of both a Domain-Specific Language (DSL), with which to describe communication protocols, and a generator that produces code from the DSL programs. After writing the specification of a protocol in the DSL, an API is generated, mapping protocol states to types in the host language. Through the correct usage of that API, the respective communication protocol can be implemented, ensuring properties such as deadlock-freedom and protocol fidelity (given that the protocol was specified correctly in the DSL), though none of these properties is ensured statically. There are instances of this type of implementation in several languages, such as in Scala [52], Java [19], F# [41], and Python [40]. In terms of advantages, since a user first specifies protocols to program in a DSL, these implementations are able to describe communication protocols in a more fine-grained manner than general-purpose library implementations. There are, however some disadvantages to these implementations: it's necessary for users to specify protocols in the DSL; and it's harder to make changes to already defined protocols, since the specifications need to be redone, and any changes in protocol involve changes to the generated APIs, breaking compatibility with previously written code.

3.1.3 Session Types by Leveraging (Advanced) Host Type System Features

Some implementations of session types take advantage of advanced type system features from the host language in which they are being implemented. Examples of such implementations exist in OCaml [24] (which use lenses), and Haskell [39, 42, 50] (which use monads and language extensions such as multi-parameter type clauses and functional dependencies). These implementations use the host type system features to encode tracking of channel usage and can statically ensure some properties that other implementations can only ensure at run-time, if at all. However, implementations of this kind present a clear disadvantage: since they rely on language specific features, it's generally harder to replicate them in the same capacity in other languages, with different kinds of feature sets. Furthermore, since specifications are represented on top of the through the encoding of existing host language types, error messages are often difficult to map to errors in the code.

```
⟨!int;⟩ $c fib(int n) {
    if (n == 0) {
        send($c, 0); close($c);
    } else if (n == 1) {
        send($c, 1); close($c);
    } else {
        ⟨!int;⟩ $c1 = fib(n-1);
        ⟨!int;⟩$c2 = fib(n-2);
        int f1 = recv($c1); wait($c1);
        int f2 = recv($c2); wait($c2);
        send($c, f1+f2); close($c);
    }
}

int main() {
    ⟨!int;⟩ $c = fib(10);
    int f = recv($c); wait($c);
    assert(f == 55);
    return 0;}
```

Figure 31: Concurrent Fibonacci Algorithm in CC0 [60]

3.2 Session-Typed Languages

3.2.1 Concurrent C0

Concurrent C0 [60], or CC0, is a type-safe C-like language, with session-typed communication over channels. Concurrent C0 is based on C0, an imperative programming language that is a strict, type safe, subset of C, designed for use in introductory programming courses. Unlike C, C0 provides memory safety by disallowing pointer arithmetic and casting; pointers come from a built-in *alloc* function and arrays come from a built-in *alloc_array* function, and they are not interchangeable, as is the case in C. The runtime of C0 checks pointer accesses for *NULL* pointers, and checks the validity of bounds in array accesses. Furthermore, C0 has garbage collection, so there is no need to explicitly free memory. C0 also supports dynamically checked contracts of the forms *@requires*, *@ensures*, *@assert* and *@loop_invariant*. Concurrent C0 extends C0 with the ability to create concurrent processes and channels to communicate between them, enriching the language with high-level concurrent programming constructs. Concurrent C0 extends C0's syntax with concurrent features. As an example of the syntax of Concurrent C0, Figure 31 shows a possible implementation of a concurrent Fibonacci algorithm.

3.2.1.1 Typed Concurrency in Concurrent C0

By adopting a session-typed approach to its typing system, Concurrent C0 benefits from all the properties of session-typing, such as expressiveness of high-level reciprocal communication patterns and session fidelity (processes send and receive correct data in accordance to channels' session types). Processes can create channels, written as $\$c$, and spawn other processes, called providers, that will send some data through those channels, to be consumed by the spawning processes (or spawning functions), called clients. Types are expressed as a semi-colon separated sequence of communications inside angle brackets. Sending and receiving are written with ! and ?, respectively. So, a channel on which a program sends a *bool* and receives an *int* would be written $\langle !bool; ?int; \rangle$. In order to fully express complex session types, Concurrent C0 introduces a choice mechanism to denote session types that branch into different sequences of types. Choices are declared in a manner similar to *structs*, as a list of labels preceded by types. Branches are selected by sending or receiving labels.

Channel variables in Concurrent C0 have a linear semantics, but with two references. Exactly one client and one provider have a reference to a channel. This ensures that communication is always one-to-one, there is never a "dangling" channel with no one listening in one of the ends, and there is never contention between multiple providers or clients to communicate in one direction over a channel. Since a provider can only have one client at a time, there is a correspondence between the client-provider relation in a Concurrent C0 program and the parent-child relationship in a tree. The *main()* function is a process with no clients and therefore the root of a tree. Concurrent C0 offers two primitives, used to ensure the linearity of the type system: *close* and *wait*. A process providing on channel $\$c$ must call close($\$c$) to terminate. The provider must have already consumed all of its references and be a leaf in the process tree - ensures every linear resource is used. Before terminating, a client listening on channel $\$c$ must call wait($\$c$) to ensure that its respective provider terminates. Channel references can be manipulated like any other variables, but are subject to linearity. They can not be copied, only renamed, and the old reference becomes unusable. When passing a channel into a spawning function, the caller gives up its reference to allow the new process to make use of the channel. Linearity ensures that channel references are not duplicated and used more than once.

Concurrent C0 also offers a forwarding operation, that allows a process with only one child in the tree to be suppressed in the communication between its parent and its child. The channel to the parent and the channel to the child are effectively merged, but session fidelity is always ensured, since the merging only happens if both channels are of the same type.

3.2.1.2 Implementation

Concurrent C0 enforces linearity and session fidelity to produce safe concurrent code. The compiler typechecks programs to make sure that messages are transmitted according to the appropriate session types, and channel linearity is respected. After the typechecking phase, the compiler adds annotations to inform the runtime about the communication structure. Lastly, the code is compiled to a target language, such as C or Go, then linked with a runtime implementation written in the same language.

Additionally, the Concurrent C0 compiler takes advantage of the characteristics of session typing to make a more efficient use of memory. For context, in Concurrent C0, a process is a unit of concurrent execution, and channels are (effectively) unbounded message buffers that allow for processes on either end to communicate asynchronously. Realistically, these message buffers take up memory space and they may need to be resized if several values need to be buffered at the same time (in the case of channels with no constraints). However, with session types, it is known the maximum amount of values that can be buffered at a time - *the type width*. For instance, the type $\langle !bool; ?int; \rangle$, which represents a channel from a server that sends a *bool* to a client, and receives an *int* as a response. A channel of this type could only ever buffers one value at a time, since the client can only send an *int*, i.e. place it in buffer, after receiving the *bool*, i.e. removing it from buffer. The compiler infers type widths, allowing the runtime to use small, fixed length buffers, without ever having to worry about resizing them.

3.2.1.3 Runtime

The Concurrent C0 compiler generates C or Go code that is linked with one of several available runtime systems that contains the logic for message-passing and process manipulating. The various runtimes differ in threading models and synchronization strategies but they share the same general structure centered around channels: a channel contains a message queue, a communication direction, a mutex, and a condition variable. The mutex is used to protect channel state, and the condition variable is used by receivers to wait on messages to arrive or the queue to change communication direction. The runtime consists of four main functions that provide all the necessary functionalities for spawning processes and passing messages; these functions are *NewChannel*, *Send*, *Recv*, and *Forward*.

The Concurrent C0 primitives *close* and *wait* are implemented by sending a special *DONE* message. *NewChannel* creates a new concurrent process and the channel along which it will provide, returning a reference to that channel to the client. *NewChannel* takes in the function and arguments for the provider process, as well as the type width and initial direction of the channel as inferred by the compiler, allowing the runtime to create a channel as a bounded buffer. *Send* is used to send a message over a given channel, additionally taking in the message's type and the inferred direction. *Send* locks the channel, enqueues the message with its type, sets the direction of the queue, and then

unlocks. The sender also signals the condition variable, since a receiver may be waiting for the message. *Recv* is used to receive a message over a given channel, taking in the message's expected type and the inferred direction. *Recv* locks the channel and attempts to receive the message. If the buffer is empty or still flowing in the opposite direction, then the receiver will give up the lock and wait for the signaling of the condition variable by the sender. If the message is a forward, the receiver handles it, installing the new channel. *Recv* asserts that the received message is of the expected type, aborting in error if it does not match. *Forward* first takes in two channels involved in a forwarding operation, along with the inferred direction of communication, then sends a a forward message in the inferred direction, and finally terminates.

3.2.1.4 Further Considerations

Since Concurrent C0 is a C-like programming language, it's missing several useful features, such as polymorphism and higher-order functions. Furthermore, it's an imperative language, different from the usual session type implementations, which opt for a more "traditional" functional approach, that more closely resembles the process calculi syntax in which session types were first described.

Chapter 4

Proposal

This chapter describes the proposal of this thesis. What we propose to do is to design a general-purpose programming language that implements logical session types. Besides the overall language design, we propose to implement several tools such as a type checker, an interpreter, and a compiler for the proposed language, along with a specific runtime to realize the concurrent operations. We also propose several possible features with which to extend the proposed language.

4.1 Language

The language we propose, based on existing language designs [54, 57], is a language designed to model the concurrent communication between processes, from the perspective of dual intuitionistic logic, aiming to ensure several desirable properties, such as deadlock-freedom, session fidelity, and progress, while at the same time ensuring its usability from a practical standpoint. We intend for the proposed language to implement the following features: session typed concurrency, recursion and general higher-order functions. The initial type syntax and process syntax for the proposed language appear in Figures 41 and 42, respectively (bound variables and channels appear in subscript). The proposed language includes *ordinary functional types*, such as numbers, booleans, recursive types, and inductive types, as well as the *usual functional constructs*, including recursion, arithmetic operations, simple polymorphism, higher-order functions and pattern matching.

There are some "standard" differences between the syntax presented in Figures 41 and 42 and the actual syntax of the language that will be used in practice, on account of the awkwardness of programming with such a mathematical syntax. For instance, instead of $\lambda x : \tau.M_x$ we would write $myFunction = fun(\tau : x)\{M_x\}$; and instead of $\otimes$, we would write $*$; instead of $\multimap$, we would write =>, among other such differences. Furthermore,

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

Figure 41: Proposed Type Syntax

M, N	::=	$\lambda x : \tau.M_x \mid MN \mid \ldots$	
	$\mid$	$a \leftarrow \{P_{a,\bar{a_i}}\} \leftarrow a_1, \ldots, a_n$	process providing a using $a_1, \ldots, a_n$
P, Q	::=	$a = Q_a; P_a$	spawn process Q_a in parallel with P_a communicating along fresh channel a
	$\mid$	$x = c.\text{recv}; P_x$	receive value/channel x along channel c and proceed as P_x
	$\mid$	$c.\text{send}x; P_x$	send value/channel x along channel c and proceed as P_x
	$\mid$	$c.\text{wait}; P$	wait for termination of channel c and proceed as P
	$\mid$	$c.\text{close};$	close channel c and terminate
	$\mid$	case c of $l_1 : P_1, \ldots, l_n : P_n$	branch on selection of l_j along channel c
	$\mid$	$c.l_j; P$	select label l_j along channel c
	$\mid$	$\text{fwd}(c_1, c_2)$	forward between c_1 and c_2
	$\mid$	if M then P else Q	conditional on value M

Figure 42: Proposed Process Syntax

in the concrete proposed language, the definition of recursion is nominal, versus the theory-friendly fixed-point version presented in Figure 41.

Note that the presented syntax in Figure 42 does not specify explicit channel creation; instead channels are created implicitly when another process is spawned. This choice to "hide" channel creation has the benefit of simplifying language syntax due to technical benefits detailed in previous work [54].

One aspect of note in the implementation of a type system pertains to "subtyping" in the presence of recursive types [48]. Suppose, for instance we define the types $T_1 = \nu X.int \wedge int \wedge X$ and $T_2 = \nu X.int \wedge X$. Both of these types are essentially the same, since both of them can unfold infinitely to represent "the sending of an infinite number of integers", so they both must be able to type the same processes. As an example, take the following program:

```
{c:T} nats(int n) {
    c.send n
    c.send n+1
    c = nats(n+2)
}
```

When implementing the type system, we must take care that T be equal to T_1 or to T_2.

4.1.1 Language Examples

A program in our proposed language is comprised of the following: *type declarations*, *function declarations and definitions*, and a special function, *main* where the execution of a program starts. Note that *main* has the special type $\top$, reserved only for it, meaning it does not need to consume all outstanding resources before terminating. The reason for this exceptional typing is that it allows for the possibility of interaction with processes that do not terminate. The typing rule for $\top$ is the following:

$$(\top)\frac{}{\Delta \vdash \top}$$

Figures 43 and 44 show examples of complete programs in the proposed language. Figure 43 shows a simple implementation of the recursive Fibonacci algorithm in the proposed language. Figure 44 shows the interaction between a *client* that and a banking system, made up of an authentication server *authServer* responsible for authenticating the users of the system and a database server *dbServer*, where the balances of all the users of the system are stored. The *client* wishes to know its balance; to do this, the *client* first sends its *password* to the *authServer*; then, the *authServer* validates the password and returns a channel for the *dbServer* if the password is valid; the *client* sends the *dbServer* its *account number*; the *dbServer* returns the balance of the *client*. If the *client* sends the wrong password, the *authServer* closes its connection.

4.2 Type Checker

An essential part of a typed-language is the type checker, which runs at compile-time to ensure programs are correctly typed. Since a *well-typed* program is free of type violations, then a program that is deemed correctly typed by the type checker will not yield type-related errors at run-time - as such there is no need to keep track of typings at run-time checking, (often) leading to more efficient compiled code [37]. Type checking is generally framed between two polar extremes: *type inference*, for languages which have no type annotations, except when absolutely necessary (e.g. module interfaces); and *strict typechecking* for languages in which every variable declaration has a type annotation (as is the case for most mainstream languages). There are advantages to both approaches: the presence of type annotations helps in the process of type checking, and serves as a

```
FibType = int & 1;

{c:FibType} fib(int x) {
    if x == 0 then
        c.send x
        c.close
    else if x == 1 then
        c.send x
        c.close
    else
        a = fib(x-1)
        b = fib(x-2)
        y = a.recv
        a.wait

        z = b.recv
        b.wait
        c.send(y+z)
        c.close
}

{c:Top} main() {
    d = fib(5)
    x = d.recv
    d.wait
    // output x
    c.close
}
```

Figure 43: Concurrent Fibonacci

form of documentation of the intent of the programmer, however having to write type annotations constantly can become very onerous for the user of a language.

The type inference approach presents a key problem, however: type inference is undecidable for most expressive type systems. To overcome the limitation of classical type inference systems, *bidirectional* type checking algorithms were developed [10]. Bidirectional type algorithms [5, 9] work differently from classical type inference; instead of trying to infer the type of all expressions in a program, knowing only the types of variables, bidirectional algorithms alternate *synthesizing*, or inferring, types, and *checking* types against those already known. In terms of *type judgments*, instead of having $\Gamma \vdash e : \tau$ (under assumptions in the context Γ, the expression e has type τ), bidirectional typing systems have two kinds of type judgments: $\Gamma \vdash e \Rightarrow \tau$ (under assumptions in the context Γ, the expression e synthesizes type τ); $\Gamma \vdash e \Leftarrow \tau$ (under assumptions in the context Γ, the expression e checks against type τ). These two judgments, called *inference judgment* and *checking judgment*, respectively, differ in which of their parts are the *inputs*, and which

```
AuthType = int >> or{ok:(DbType) * 1, nok: 1};
DbType = int >> int & 1;
ClientType = or{ok:int & 1, nok: 1};

{c:AuthType} authServer() {
    pwd = c.recv
    if checkValid(pwd) then
        c.ok
        d = dbServer();
        c.send d
        c.close
    else
        c.nok
        c.close
}
{c:DbType} dbServer() {
    acc = c.recv
    c.send queryBalance(acc)
    c.close
}
{c:ClientType} client(int acc, int pwd, {AuthType} server) {
    // d = authServer()
    server.send pwd
    case server of
        ok:
            c.ok
            server.send acc
            bal = server.recv
            c.send bal
            c.close
        nok:
            c.nok
            c.close
}
{c:Top} main() {
    password = 01234
    account = 56789
    server = authServer()
    d = client(account, password, server)
    case d of
        ok:
            x = d.recv
            // output x
            c.close
        nok:
            c.close
}
```

Figure 44: Banking System Interaction

$$(T-APP)\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1 e_2 \Rightarrow \tau_2} \qquad (T-FN)\frac{\Gamma, x:\tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.e \Leftarrow \tau_1 \to \tau_2}$$

Figure 45: Bidirectional Type Checking Rules for Functions

parts are the *outputs*. For instance, when deriving $\Gamma \vdash e \Rightarrow \tau$, only Γ and e are known, and the type τ must be inferred from e - e is input and τ is output. And when deriving $\Gamma \vdash e \Leftarrow \tau$, τ and Γ are known, and we need to check $e$against τ - τ is input and e is output. From these two judgments, we can extract two mutually recursive functions to use algorithmically, an inference function, and a checking function. The checking function calls the inference function when it reaches an expression whose type can be inferred; and the inference function calls the checking function when it encounters a type annotation.

As an example of the typing rules of a bidirectional type checking system, Figure 45 shows the typing rules for function application, *T-APP* and declaration, *T-FN*. A function application $e_1 e_2$ synthesizes the type τ_2 under Γ, if function e_1 synthesizes type τ_1 under Γ and function arguments e_2 can be checked against type τ_2 under Γ. A function declaration $\lambda x.e$ checks against type $\tau_1 \to \tau_2$ under Γ if function result checks against type τ_2 under $\Gamma, x:\tau$.

These algorithms scale to extremely powerful type systems [11], are relatively simple to implement, and typically tend to produce error messages that are "local", i.e. near the specific level where problems occur. With these bidirectional type checking algorithms, type annotations are only necessary for polymorphic reducible expressions [10].

One key aspect of checking session-typed programs is linear resource management. Previous rules are not algorithmic since they "hide" context management i.e. they do not specify how linear hypotheses should be split between the two premises. There is, however, a way to manage linear context using linear logical programming techniques [47, 48]: instead of trying to guess how linear hypotheses should be split between two premises of a rule, all linear variables are passed to the first premise. Checking the corresponding subterm will consume some of these variables, and the remaining variables are passed to check the second subterm. This technique requires a judgment of the form

$$\Gamma; \Delta_I \setminus \Delta_O \vdash M : A$$

where Δ_I represents the available linear hypotheses and $\Delta_O \subseteq \Delta_I$ represents the linear hypotheses not used in M. As an example, the $\otimes R$ rule in dual intuitionistic linear logic:

$$(\otimes R)\frac{\Gamma; \Delta \vdash D : A \qquad \Gamma; \Delta' \vdash E : B}{\Gamma; \Delta, \Delta' \vdash \otimes R\ D\ E : A \otimes B}$$

could be written in the following way using context management techniques:

$$(\otimes R)\frac{\Gamma; \Delta_I \setminus \Delta \vdash D : A \qquad \Gamma; \Delta \setminus \Delta_O \vdash E : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash \otimes R\ D\ E : A \otimes B}$$

This technique of managing linear context is compatible with bidirectional formulations. In fact, this rule could be written in the following way:

$$(\otimes R)\frac{\Gamma; \Delta_I \setminus \Delta \vdash P :: y \Rightarrow A \qquad \Gamma; \Delta \setminus \Delta_O \vdash Q :: x \Rightarrow B}{\Gamma; \Delta_I \setminus \Delta_O \vdash (\nu y) x\langle y \rangle.(P \mid Q) :: x \Rightarrow A \otimes B}$$

Thus, we propose to implement a bidirectional type checker for our proposed language.

4.3 Interpreter

We propose to implement an interpreter for our proposed language. The main point of debate when choosing what kind of interpreter to implement is the type of concurrency that the interpreter will support: *"true" concurrency* - using concurrent constructs, such as threads, to simulate processes, versus *simulated concurrency* - simulating all processes in a single thread. Of these two choices, "true" concurrency is more attractive, since it more accurately reflects the reality of communications, an so it's the desirable form of concurrency to choose for our interpreter. If we implement our interpreter from scratch, then, in order for it to support multiple threads, we will also need to design and implement a scheduler, which entails a great deal of programming and testing effort. There is also the possibility of implementing our interpreter in an existing language; in that case we could make use of the host language's concurrent features implement "true" concurrency. We propose to study the possible features of our interpreter, including the kind of concurrency it will support, and the language in which it will be implemented.

4.4 Compiler

We propose to implement a compiler for our proposed language. One of the main concerns when implementing a compiler is choosing the desired *compilation target*; a somewhat difficult choice, given the plethora of existing compilation targets each with its own advantages and disadvantages. We could compile our proposed language to a *virtual machine*, such as the JVM, the LLVM, or the .NET Framework. A clear advantage is the portability of such a solution. Some virtual machines offer useful features such as garbage collection, which free us from having to manage memory "by hand". There is, however, the possibility that compiling and running our code in a virtual machine could incur a performance cost due to the virtual machine's overhead. There is also the possibility of compiling our proposed language to the assembly code, or even machine code, of a specific *hardware architecture*. With this solution, there is the opportunity for a compilation that is more tailored to the needs of our proposed language, resulting in increased performance; however, portability is severely limited, and compilation to a target at this low-level of abstraction is very hard and error prone. Finally, we could *transpile* our proposed language into another general-purpose language, which would then be

compiled/interpreted by its specific compiler/interpreter. Since we're translating into a general-purpose language, we have access to all of its high-level constructs, which could make the implementation of the compiler (specifically transpiler) much easier; however, it's also possible that the features of the transpilation target do not adequately meet all the needs of our proposed language, leading to overly complicated transpiling solutions.

We propose to study the existing compilation targets shown above, in order to choose the most appropriate compilation target for our proposed language. In order to make this choice, we will take into account the example of Concurrent C0 [60], in which compiler takes advantage of the characteristics of session types to make a more efficient use of memory.

4.5 Runtime

We propose to develop a runtime for managing the logic of the session type implementation for the proposed language. Compiled code will be linked with the runtime. We propose to study possible languages in which to implement our proposed runtime. We will draw inspiration from Concurrent C0 [60] to implement the proposed runtime, specifically, we propose to implement communication channels as on-demand memory buffers.

4.6 Extensions

The proposed language could be extended with several features. However, we propose to extend our language with two concurrency specific features, refined session types (a lightweight form of type dependency), and "unsafe" code interaction points, since their potential increase to the proposed language's expressiveness outweighs their costs of implementation.

4.7 Evaluation

We will need to validate the correctness of our approaches, from the implementation of the type checker, to the implementation of the interpreter and the runtime, to the implementation of the compiler. We propose to study the best way to test the various approaches, and write a battery of tests for each of them in order to validate their correctness. We will also illustrate the expressiveness of our design with various case studies.

4.8 Timeline

Figure 46 displays the proposed work plan. We will start by implementing and then validating the language type checker, without which the development of the rest of the tools can not proceed, since the type system and its correctness is of utmost importance for this language. Then we will implement the language interpreter, according to whichever

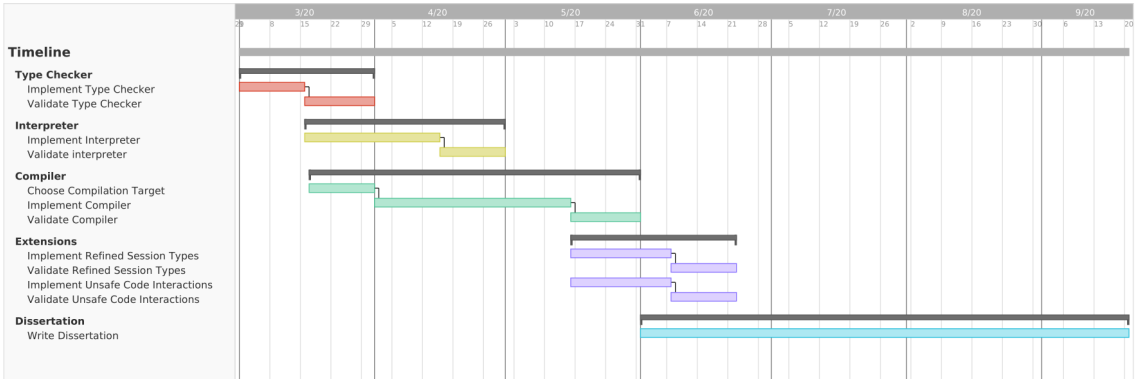

Figure 46: Planned Work Schedule

concurrency model we deem acceptable, and decide on a suitable compilation target. Afterwards, we will validate the interpreter and implement the compiler - this will be the most complex implementation task, and as such will take most of the time. After that, we will validate the compiler through testing, and start implementation of the proposed language extensions: refined session types and unsafe code interactions. Finally, we will validate the implemented extensions and write the complete dissertation.

Bibliography

[1] G. R. Andrews. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991. ISBN: 978-0-8053-0086-4.

[2] S. Balzer, B. Toninho, and F. Pfenning. "Manifest Deadlock-Freedom for Shared Session Types." In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. 2019, pp. 611–639. DOI: 10.1007/978-3-030-17184-1_22. URL: https://doi.org/10.1007/978-3-030-17184-1_22.

[3] L. Caires and F. Pfenning. "Session Types as Intuitionistic Linear Propositions." In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. 2010, pp. 222–236. DOI: 10.1007/978-3-642-15375-4_16. URL: https://doi.org/10.1007/978-3-642-15375-4_16.

[4] L. Caires, F. Pfenning, and B. Toninho. "Towards concurrent type theory." In: *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*. 2012, pp. 1–12. DOI: 10.1145/2103786.2103788. URL: https://doi.org/10.1145/2103786.2103788.

[5] D. R. Christiansen. *Bidirectional Typing Rules: A Tutorial*. Tech. rep. Max Planck Institute for Software Systems, 2013.

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. "Links: Web Programming Without Tiers." In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. 2006, pp. 266–296. DOI: 10.1007/978-3-540-74792-5_12. URL: https://doi.org/10.1007/978-3-540-74792-5_12.

[7] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. "Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication." In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. 2012, pp. 228–242. DOI: 10.4230/LIPIcs.CSL.2012.228. URL: https://doi.org/10.4230/LIPIcs.CSL.2012.228.

[8] A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134190440.

[9] J. Dunfield. *Bidirectional typechecking*. Tech. rep. McGill Institution, 2012.

[10] J. Dunfield and N. R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism." In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 429–442. DOI: 10.1145/2500365.2500582. URL: https://doi.org/10.1145/2500365.2500582.

[11] J. Dunfield and N. R. Krishnaswami. "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types." In: *PACMPL* 3.POPL (2019), 9:1–9:28. DOI: 10.1145/3290322. URL: https://doi.org/10.1145/3290322.

[12] S. Fowler, S. Lindley, J. G. Morris, and S. Decova. "Exceptional asynchronous session types: session types without tiers." In: *PACMPL* 3.POPL (2019), 28:1–28:29. DOI: 10.1145/3290341. URL: https://doi.org/10.1145/3290341.

[13] E. Giachino, N. Kobayashi, and C. Laneve. "Deadlock Analysis of Unbounded Process Networks." In: *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. 2014, pp. 63–77. DOI: 10.1007/978-3-662-44584-6_6. URL: https://doi.org/10.1007/978-3-662-44584-6_6.

[14] M. Giunti and V. T. Vasconcelos. "A Linear Account of Session Types in the Pi Calculus." In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. 2010, pp. 432–446. DOI: 10.1007/978-3-642-15375-4_30. URL: https://doi.org/10.1007/978-3-642-15375-4_30.

[15] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic." In: 1969.

[16] K. Honda. "Types for Dyadic Interaction." In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35. URL: https://doi.org/10.1007/3-540-57208-2_35.

[17] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming." In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. 1998, pp. 122–138. DOI: 10.1007/BFb0053567. URL: https://doi.org/10.1007/BFb0053567.

[18] 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.

[19] 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.

[20] A. Igarashi and N. Kobayashi. "Type Reconstruction for Linear -Calculus with I/O Subtyping." In: *Inf. Comput.* 161.1 (2000), pp. 1–44. DOI: 10.1006/inco.2000.2872. URL: https://doi.org/10.1006/inco.2000.2872.

[21] A. Igarashi and N. Kobayashi. "A generic type system for the Pi-calculus." In: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. 2001, pp. 128–141. DOI: 10.1145/360204.360215. URL: https://doi.org/10.1145/360204.360215.

[22] A. Igarashi and N. Kobayashi. "A generic type system for the Pi-calculus." In: *Theor. Comput. Sci.* 311.1-3 (2004), pp. 121–163. DOI: 10.1016/S0304-3975(03)00325-6. URL: https://doi.org/10.1016/S0304-3975(03)00325-6.

[23] A. Igarashi, P. Thiemann, V. T. Vasconcelos, and P. Wadler. "Gradual session types." In: *PACMPL* 1.ICFP (2017), 38:1–38:28. DOI: 10.1145/3110282. URL: https://doi.org/10.1145/3110282.

[24] 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.

[25] M. Jones. *What really happened on Mars Rover Pathfinder*. 1997. URL: http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html (visited on 2020).

[26] 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.

[27] 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.

[28] N. Kobayashi. "A New Type System for Deadlock-Free Processes." In: *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*. 2006, pp. 233–247. DOI: 10.1007/11817949_16. URL: https://doi.org/10.1007/11817949_16.

[29] N. Kobayashi and C. Laneve. "Deadlock analysis of unbounded process networks." In: *Inf. Comput.* 252 (2017), pp. 48–70. DOI: 10.1016/j.ic.2016.03.004. URL: https://doi.org/10.1016/j.ic.2016.03.004.

[30] 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.

[31] N. Kobayashi, B. C. Pierce, and D. N. Turner. "Linearity and the Pi-Calculus." In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 1996, pp. 358–371. DOI: 10.1145/237721.237804. URL: https://doi.org/10.1145/237721.237804.

[32] N. Kobayashi, B. C. Pierce, and D. N. Turner. "Linearity and the pi-calculus." In: *ACM Trans. Program. Lang. Syst.* 21.5 (1999), pp. 914–947. DOI: 10.1145/330249.330251. URL: https://doi.org/10.1145/330249.330251.

[33] N. G. Leveson. "The Therac-25: 30 Years Later." In: *Computer* 50.11 (2017), pp. 8–11. ISSN: 1558-0814. DOI: 10.1109/MC.2017.4041349.

[34] 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.

[35] S. Lindley and J. G. Morris. "12 Lightweight Functional Session Types." In: 2017.

[36] S. Marlow. *Haskell 2010 Language Report*. 2010.

[37] R. Milner. "A Theory of Type Polymorphism in Programming." In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[38] R. Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3. URL: https://doi.org/10.1007/3-540-10235-3.

[39] 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.

[40] 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.

[41] R. Neykova, R. Hu, N. Yoshida, and F. Abdeljallal. "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.

[42] 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.

[43] L. Padovani. "Deadlock and lock freedom in the linear π-calculus." In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. 2014, 72:1–72:10. DOI: 10.1145/2603088.2603116. URL: https://doi.org/10.1145/2603088.2603116.

[44] 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.

[45] L. Padovani and L. Novara. "Types for Deadlock-Free Higher-Order Programs." In: *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. 2015, pp. 3–18. DOI: 10.1007/978-3-319-19195-9_1. URL: https://doi.org/10.1007/978-3-319-19195-9_1.

[46] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. "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.

[47] F. Pfenning. *Linear Type Checking*. Tech. rep. Carnegie Mellon University, 2001.

[48] F. Pfenning and D. Griffith. "Polarized Substructural Session Types." In: *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 3–22.

DOI: 10.1007/978-3-662-46678-0_1. URL: https://doi.org/10.1007/978-3-662-46678-0_1.

[49] K. Poulsen. *Software Bug Contributed to Blackout*. 2004. URL: https://www.securityfocus.com/news/8016 (visited on 2020).

[50] 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.

[51] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001. ISBN: 978-0-521-78177-0.

[52] 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.

[53] E. Sumii and N. Kobayashi. "A Generalized Deadlock-Free Process Calculus." In: *Electr. Notes Theor. Comput. Sci.* 16.3 (1998), pp. 225–247. DOI: 10.1016/S1571-0661(04)00144-6. URL: https://doi.org/10.1016/S1571-0661(04)00144-6.

[54] B. Toninho. "A Logical Foundation for Session-based Concurrent Computation." In: 2015.

[55] B. Toninho and N. Yoshida. "Depending on Session-Typed Processes." In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 2018, pp. 128–145. DOI: 10.1007/978-3-319-89366-2_7. URL: https://doi.org/10.1007/978-3-319-89366-2_7.

[56] B. Toninho, L. Caires, and F. Pfenning. "Dependent session types via intuitionistic linear type theory." In: *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*. 2011, pp. 161–172. DOI: 10.1145/2003476.2003499. URL: https://doi.org/10.1145/2003476.2003499.

[57] B. Toninho, L. Caires, and F. Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration." In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 350–369. DOI: 10.1007/978-3-642-37036-6_20. URL: https://doi.org/10.1007/978-3-642-37036-6_20.

[58] J. D. Ullman. *Elements of ML Programming*. USA: Prentice-Hall, Inc., 1994. ISBN: 0131848542.

[59] P. Wadler. "Propositions as sessions." In: *J. Funct. Program.* 24.2-3 (2014), pp. 384–418. DOI: 10.1017/S095679681400001X. URL: https://doi.org/10.1017/S095679681400001X.

[60] 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.