Mini-EnsembleS: Typechecking a Core Session-Typed Adaptive Actor Language

Himanshu Chopra

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

December 15, 2021

Abstract

Technology is evolving faster than ever as each technological improvement creates a stronger generation of technologies at a faster rate, leading to the rapid growth of heterogeneous hardware devices and distributed systems. Software systems and applications must be able to adapt to their execution surroundings to make the best use of their resources.

Dynamic self-adaptation is a systems concept that allows devices to discover, replace and communicate with other software components at run-time. Runtime discovery and replacement of software components can be utilized by many applications to sense and adapt to their environments. The shared-nothing semantics and explicit message passing of the actor model of computation provide a pertinent structure of programming such adaptive systems. Ensemble is an imperative actor-based concurrent programming language that supports software adaptation. However, Ensemble is vulnerable to concurrency errors such as deadlocks and communication mismatches. Multiparty session types (MPST) are used to describe communication protocols involving any number of participants, preventing errors and guaranteeing their correctness. This is achieved by ensuring each actor conforms to a predefined communication protocol and that the protocol is well-formed and valid according to MPST theory. Extending MPST theory with explicit connection actions allows flexible communication protocols with optional and dynamic participants.

EnsembleS is a new session-typed actor-based language based on Ensemble, introducing the concepts of MPST with explicit connection actions to support compile-time verification of safe runtime adaptation. The current implementation of EnsembleS is built on top of Ensemble; this introduces implementation artefacts and is more difficult to extend and relate to the formal model. The project aims to implement a type-checker for a smaller core language based on the formal model of EnsembleS, which can then be used for experimenting and further extensions. The implementation is evaluated by applying it to different case studies such as a Ping-Pong scheme, a Bookstore, and an adaptive DNS server.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Himanshu Chopra Signature: H.Chopra

Acknowledgements

I would like to thank my supervisor, Dr. Simon Fowler, for his support and guidance throughout the course of this project. His enthusiasm and passion for the theory of programming languages and complex computational systems is infectious and piqued my interest, enabling me to pursue this particular project.
I would also like to thank my mother, Lata Chopra, who has always supported and believed in me, and my sister, Kajol Chopra, for her moral support.

Contents

1 Introduction **5**

2 Background **7**

2.1 Survey 7

2.1.1 The Actor Model of Concurrency 7

2.1.2 Software Adaptation 9

2.1.3 Multiparty Session Types with explicit connection actions 9

2.1.4 EnsembleS 10

2.2 Calculus 11

2.2.1 Syntax 11

2.2.2 Typing Rules 13

2.2.3 Term Typing 14

2.3 Design 16

2.3.1 Parser 17

2.3.2 Type System 17

3 Implementation **18**

3.1 Abstract Syntax Tree 18

3.2 Parser 19

3.2.1 Parsing environment 19

3.2.2 Running the parser 20

3.3 Type Checker 20

3.3.1 The Either type 20
3.3.2 The custom Error data type 21
3.3.3 Typing Environment and State 22
3.3.4 Equi-recursive Session types 23
3.3.5 Structure of Type System 23
3.3.6 Executing Type-Checker 23

4 Evaluation **26**

4.1 Ping-Pong 27
4.2 Online Store 29
4.3 DNS server 31

5 Conclusion and Future Work **33**

A Appendix **34**

A.1 Concrete Syntax 34
A.1.1 Concrete Syntax for Types and Terms 35
A.1.2 Concrete Syntax for Session action and Session types 35
A.2 Abstract Syntax Tree 36

Chapter 1

Introduction

Technology is evolving at an accelerating pace, creating a new, stronger generation of technologies. Computing platforms have evolved from isolated computers to heterogeneous hardware devices, consisting of multicore CPUs and co-processors, to embedded IoT devices and self-driving cars. Due to the wide diversity of operating conditions connected with these platforms and the different resource constraints in these environments, applications must be able to adapt to their execution environment to make the greatest use of the resources available. Applications must be able to successfully operate in different environments, identify resources, and efficiently communicate with other technologies. Communication in these distributed systems is becoming one of the central elements in the software development of these technologies [12]. This is especially challenging as different classes of hardware devices each have different programming styles and runtimes, requiring developers to have knowledge of different programming styles, hardware platforms, and networking protocols [10].

Dynamic self-adaptation is a systems concept that allows devices to discover, connect and communicate with other software components at run-time [11]. Instead of modifying entire applications, adaptation provides software components with the ability to discover and communicate with other components which were not part of the original system design. Adaptation also involves replacement of executing components to update, swap or extend functionalities of the existing software. Communication protocols in distributed systems are quite complex; the order of communication and the type of data exchanged is dictated by these protocols. Most mainstream programming languages today do not support mechanisms to ensure that dynamic adaptation involving these communication protocols can be safely achieved or result in rather prohibitive runtime overheads [11].

In the actor model of adaptation, actors have shared-nothing semantics i.e. they share no state and interact using explicit message passing. This model provides the ideal structure to facilitate runtime adaptation [10]. Ensemble is an imperative actor-based language that has native support for adaptation [11]. However, using Ensemble for runtime-adaptation can lead to concurrency errors, deadlocks amongst actors and communication mismatches and hence, cannot guarantee safety [11]. Multiparty session types can be used to address this problem.

Session types have been researched in detail as a potential typed foundation for structured communication centred programming [12, 21]. Communication-based applications frequently display a highly ordered sequence of interactions that, when taken together, can be expressed as a session unit. A session's structure is abstracted as a type, which is then used to validate applications through an

associated type discipline [20]. Session types thus provide a typing discipline by assigning session types to communication channels, and validating them to ensure communication safety.

As a simple example, the following session type describes a communication protocol between a Customer and an Agency from the Customer's perspective i.e., a local view of the binary session. In this protocol, the Customer sends the selected travel location (a string), the Agency sends a price quote for that location (an integer). The customer makes a decision/choice at this point, to either accept the offer or reject and terminate the communication. If satisfied by the quote, the Customer sends his address (a string) and the Agency sends the confirmed dates of travel (a date) and the protocol terminates.

```
! string . ? int . (! string . ? date . end) + (end)
```

'!' denotes a send action, '?' denotes a receive operation and '.' is a sequencing operator. Expressing protocols using these explicit actions help deal with synchronisation as well as allow validation of communication safety through the exchange of messages.

Session types, initially only able to describe communications between two ends of a channel, are later extended to multiparty [12], giving rise to the multiparty session types (MPST) theory. MPST generalise the binary session type theory to the multiparty case, ensuring that protocols, involving two or more participants, are free from communication errors and deadlocks if the processes are well-typed [21]. The core idea of the formalism is to specify the type, direction and sequence of communication actions by introducing global types. Global types describe the conversations from a global perspective and provide a tool to check protocol agreement. The approach is as follows:

1. Construct a global type that provides a shared contract for the systems' acceptable message exchange pattern.

2. Project the global type to each participating actor to get a local type, which describes the protocol from the local point of view.

3. Check that each process' implementation adheres to its local type.

These local types are used to validate an application through type-checking. The local conformance of each participant guarantees global conformance of the network [20].

EnsembleS is an actor-based language that integrates the concepts of MPSTs with explicit connection actions [13] to provide compile-time verification of safe adaptation [11]. It integrates the notions of explicit connection actions and type-checking of MPST to the actor paradigm, which allows to statically guarantee that a discovered actor will follow the communication protocol and that changing the actor's behaviour will not threaten the communication safety. EnsembleS is implemented directly on top of Ensemble, which introduces various implementation artefacts and obscures the core language's fundamental concepts. § 4.3 discusses the peripheral code issues in a communication scheme expressed in EnsembleS [11]. The calculus of EnsembleS is minimal and can be implemented using a much smaller core language. This project aims to implement a typechecker for a small, clean core language based on this calculus, which conveys the core ideas better. The implementation is evaluated by applying it to different case studies such as a Ping-Pong scheme, an Online-Store example, and an adaptive DNS server. This model provides a more flexible and extensible tool for future extensions and experiments.

Chapter 2

Background

2.1 Survey

2.1.1 The Actor Model of Concurrency

Hardware technology is evolving rapidly as faster chips are replaced by an ever-increasing number of cores. As a result, computing platforms have evolved into heterogeneous devices consisting of multicore CPUs and co-processors; concurrent and distributed software programming is essential in utilizing these hardware resources. Most popular languages use the shared-state concurrency model that rely on mutable shared state and locking mechanisms and make good use of multiple cores. However, synchronization issues, lock contentions, and deadlocks may occur in such programming [5]. The Actor Model of computation is a message-passing concurrency model that promises better reliability and scalability.

Actors, as described in the Actor model, are lightweight user-process that share no state and communicate by asynchronous message passing. A message can be a simple string [2], or can consist of an identifier that defines the type of message and a payload [11]. The order of processing of these messages may coincide with their order of arrival or may be dependent on some intrinsic property like priority or tag (selective processing)[6]. Out-of-order message processing is common in various implementations of actors such as Erlang, Scala, and Akka actors [6]. The behaviour of a particular actor is driven by these received messages, stored in the actor's mailbox.

The no-shared policy between actors and the asynchronous nature of the communication (allowing non-blocking sends and buffering of received values) between these isolated entities ensure that common concurrency issues such as low-level data races and deadlocks are avoided [5, 8]. Each of these actors can spawn a finite set of other actors, communicate using message passing and modify the behaviour of itself or other actors based on incoming messages [1]. The architecture of the Actor Model can thus, be used to facilitate concurrent programming in massive parallel workstations as well as in distributed settings [5].

Erlang was the first industry-strength language to adopt the actor model of concurrency and has seen a renewed interest due to its native support for multicore and distributed programming. It is a concurrent, functional programming language designed for programming massively scalable and

```
1  -module(mod1).
2  -export([start/0, ping/1, pong/0]).
3  ping(0) ->
4    pong ! finished,
5    io:format("ping finished~n", []);
6  ping(N) ->
7    pong ! {ping, self()},
8    receive
9      pong ->
10       io:format("Ping received pong~
       n", [])
11   end,
12   ping(N - 1).
13 pong() ->
14   receive
15     finished ->
16       io:format("Pong finished~n", []);
17     {ping, Ping_PID} ->
18       io:format("Pong received ping~n", [
       ]),
19       Ping_PID ! pong,
20       pong()
21   end.
22 start() ->
23   register(pong, spawn(mod1, pong, [])),
24   spawn(mod1, ping, [2]).
```

Figure 2.1: Erlang Message Passing

fault-tolerant distributed systems [2]. In Erlang, the notion of a process is fundamental, and these are created and managed by the Erlang runtime environment, not by the underlying operating system. The Erlang runtime environment (a virtual machine) allows a code compiled on one architecture to run anywhere. All processes are isolated from one another and do not share memory; the exchange of data is done via message passing as directed by the Actor model [1].

Figure 2.1 displays a ping-pong scheme written in Erlang. Two processes - ping & pong are created which send messages to each other for a defined number of times. The function `start` creates the processes and initiates the communication. The processes communicate using the `!` operator (lines 4,19). Each process in turn calls itself until termination. The output of the program is displayed adjacently.

```
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
```

Erlang applications are composed of hundreds of such lightweight processes and fully utilize a multicore environment. Erlang's distribution mechanisms are transparent; programs need not be aware that they are distributed, which allows designing these applications without bothering much about concurrency. Erlang also provides a powerful mechanism for error handling and fault tolerance(supervised processes) [2]. The ideology is to let failing processes crash (since this does not affect other isolated processes) and let other processes detect and fix them. Fault-tolerant applications require four key properties [2]: Isolated processes, Pure message passing between processes, the ability to discover errors in remote processes, and mechanisms to determine the cause of error and how to deal with them. Erlang modules include processes, powerful exception management strategies, code replacement mechanisms, and a large set of libraries; it is well placed for concurrent programming. Below is an example of a ping-pong scheme

Akka is a concurrency framework for building highly concurrent and distributed actor-based systems on the JVM [4]. It is available as open source and distributed through libraries implemented with the SCALA programming language. The middleware platform is based on the concept of Actor Model, with inspiration drawn from Erlang [5]. Akka hosts a hierarchical structure of actors which allows the implementation of a powerful exception handling mechanism. This feature, along with easy deployment of network distributed systems, has popularized the Akka toolkit in industrial and commercial applications.

2.1.2 Software Adaptation

As heterogeneous components and IoTs are developed, static configurations are no longer adequate as applications must learn to sense their resource environments, and communicate with other software components at runtime [10]. Datacenters with distributed machines, self-driving cars, or tablet machines with desktop performance expectations, all require to perform and adapt in dynamic environments. Dynamic self-adaptation is a systems concept that allows devices to successfully operate by sensing and responding to their surrounding environment, discovering, replacing, and communicating with other software components at runtime which were not part of the original system design [11]. This is challenging as each component may follow its own programming environment and runtime which needs to be handled individually.

RE^X is a development platform for runtime emergent software systems, which uses purely machine-driven decisions for assembling and adaptation to produce systems that are responsive to runtime environmental conditions [18]. However, the adaptive aspects have not been composed at runtime and safety cannot be guaranteed as RE^X system's programming language does not specify communication protocols for concurrent components [11]. Most mainstream programming languages do not support mechanisms to ensure that dynamic adaptation can be achieved safely or result in rather prohibitive runtime overheads [11]. Actor-based approaches, like Erlang supports remote location communication and actor creation but does not support migration (moving an executing actor to another location) or runtime discovery of resources [10]. However, extending an actor-based framework such as Ensemble enables programming of applications with runtime adaptation via discovery and replacement, and demonstrates that an actor can be used as an appropriate unit of adaptation [10, 11].

2.1.3 Multiparty Session Types with explicit connection actions

Multiparty session types (MPST) is a type systems theory that generalizes the binary session type theory for verifying message-passing concurrent processes [12]. The main design philosophy follows a top-down approach (Fig 2.2): (a) a global type describing the message-passing communication protocol from a global perspective, providing essential sequencing information, in the form of a shared contract among all participants. (b) a localized view of the protocol, obtained via projection from the global type, for each participant; which is used for (c) type-checking to ensure that participating processes follow these local types; the local conformance of each participant guarantees the global conformance of the network, guaranteeing that the system is free from communication safety errors, mismatches and deadlocks [20, 19]. Traditional MPST theory expects an atomic synchronization between all participants at session initiation, assuming an interconnected structure between a fixed number of participants (Static topology)[13]. This system does not consider a dynamic session where connections can be established, closed, and possibly re-established as the protocol progresses. Real-world applications require more practical and flexible session types; MPST can be extended to support explicit connection actions to enable a more relaxed form of MPST theory, that allows optional roles, and dynamic joining and leaving of roles (Dynamic topologies) [13].

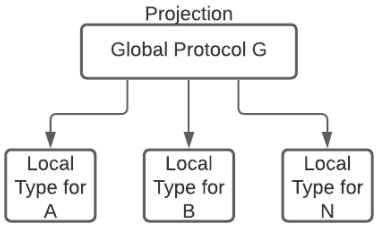


Figure 2.2: MPST workflow

```
1  global protocol Bookstore (role Sell, role Buy1,
       role Buy2) {
2    book(string) from Buy1 to Sell;
3    book(int) from Sell to Buy1;
4    quote(int) from Buy1 to Buy2;
5    choice at Buy2 {
6      agree(string) from Buy2 to Buy1, Sell;
7      transfer(int) from Buy1 to Sell;
8      transfer(int) from Buy2 to Sell;
9    } or {
10     quit(string) from Buy2 to Buy1, Sell;
11   } }
```

```
12 local protocol Bookstore_Sell (self Sell,role Buy1
       ,role Buy2) {
13   book(string) from Buy1;
14   book(int) to Buy1;
15   choice at Buy2{
16     agree(string) from Buy2;
17     transfer(int) from Buy1;
18     transfer(int) from Buy2;
19   } or {
20     quit(string) from Buy2;
21   }
22 }
```

Figure 2.3: Global and local protocols for bookstore

Scribble [22] is a protocol description language based on the theory of MPST. Using Scribble, global protocols can be validated and projected, to check the safety of these protocols. Figure 2.3 shows the Scribble code for the global and local protocols in a classic Bookstore scenario [11]. This involves two buyers Buy1, Buy2, who wish to purchase a book from the seller Sell. Buy1 sends the selected book string to Sell (line 2). Sell sends the price of the book to Buy1 (line 3) at which point Buy1 invites Buy2 to split the price (line 4). Buy2 can either agree which leads to the transfer of funds (lines 6-8) or Buy2 can quit the protocol (line 10). Projecting this global protocol into the local protocols shows actions only relevant to the corresponding participant; Bookstore_Sell is the projected local protocol for Sell.

Ensemble is an imperative actor-based language specialized for designing distributed applications [10]. It has lightweight, single-threaded processes and these processes share no state amongst one another. They interact via explicit message passing along uni-directional, typed channels (buffers), as opposed to associated mailboxes in actor models [8]. Ensemble applications are compiled in Java source code and can be executed on desktops and various hardware platforms. Extending the Ensemble language and environment allows the runtime discovery of actors, location transparent communication via channels as well as spawning and migration of actors across supported platforms; thus, owing to its design, Ensemble can support dynamic self-adaptation [10]. However, using Ensemble for runtime-adaptation can lead to concurrency errors, deadlocks amongst actors, and communication safety errors and hence, cannot guarantee safety [11]; integrating MPST with explicit connection actions can be used to address this problem.

2.1.4 EnsembleS

EnsembleS is an actor-based language that integrates the concepts of MPSTs with explicit connection actions to provide compile-time verification of safe adaptation [11]. It integrates the notions of explicit connection actions and type-checking of MPST to the actor paradigm, which allows to statically guarantee that a discovered actor will follow the communication protocol and that changing the actor's behaviour will not threaten the communication safety. EnsembleS actors have their own private state and a single thread of control expressed as a behaviour clause. EnsembleS supports traditional MPST theory with a fixed number of participants i.e., static topologies as well as dynamic topologies by adopting explicit connection actions. The core calculus of EnsembleS (§ 2.2) proves that the integration of adaptation with multiparty session types is safe.

Fig. 2.4 shows how an actor template code is generated from a defined global session type using Scribble. The input protocol is validated and projected onto local protocols; for each of these locol

Figure 2.4: EnsembleS Actor template code generation

types, the StMungo tool produces the session type, interface and actor template. The code is fed into the EnsembleS compiler to produce the executable code. EnsembleS supports runtime discovery of local or remote actors as well as replacement of executing actors to facilitate adaptation.

EnsembleS is implemented directly on top of Ensemble, which introduces various implementation artifacts; Ensemble uses interfaces and unidirectional, simply-typed channels for message passing, as opposed to mailboxes in other actor-based models. This requires additional implementation undertaking [11] and obscures the core language's fundamental concepts. The calculus of EnsembleS, defined in § 2.2, is minimal and can be implemented using a much smaller core language. This project aims to implement a type-checker for a small, clean core language based on this calculus, which conveys the core ideas better. The implementation is evaluated by applying it to different case studies chapter 4 such as a Ping-Pong scheme, an Online-Store example, and an adaptive DNS server. This model provides a more flexible and extensible tool for future extensions and experiments.

2.2 Calculus

This section introduces the syntax and semantics of the formal calculus of EnsembleS [11]. This is the core conceptualization of EnsembleS that allows for adaptation, while preventing the inherent communication mismatches owing to characteristics of Ensemble using the theory of Multiparty Session Types. The project aims at implementing a type checker for this calculus and the typing rules discussed are the most crucial part of focus thus.

With explicit connection actions, the core calculus tries to distill the essence of the interplay between adaptation and session-typed communication. The choice of a functional core calculus over an imperative calculus is to avoid dealing with imperative variable bindings that scatter and clutter the formalism. Since dynamic topologies with explicit connection actions are more crucial for adaptation, the calculus focuses on these rather than static topologies.

2.2.1 Syntax

Figure 2.5 shows the syntax of types and terms in the calculus of EnsembleS [11].

Definitions: Actor class names are defined using u; D ranges over actor definitions. Each definition follows the syntax - **actor** u **follows** S $\{M\}$. This definition specifies the actor's class name, session type, and the behavior it follows. For a provided actor definition - actor u follows S {M}, we define sessionType(u) = S and behaviour(u) = M.

Values: The calculus works in the setting of fine-grain call-by-value, where there is an explicit classification of values and computations and an explicit order of evaluation [15]. Values V, W

Syntax of Types and Terms

Actor class names	u		
Actor definitions	D	::=	**actor** u **follows** S $\{M\}$
Roles	$\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}$		
Recursion Labels	l		
Behaviours	κ	::=	M | **stop**
Types	A, B	::=	Pid(S) | string | int | bool | **1**
Values	V, W	::=	x | s | i | b | ()
Actions	L	::=	**return** V | **continue** l | **raise**
		|	$V == W$ | $V \neq W$
		|	**new** u | **self** | **replace** V **with** κ | **discover** S
		|	**connect** $\ell(V)$ **to** W **as** $\mathtt{p}$ | **accept from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**send** $\ell(V)$ **to** $\mathtt{p}$ | **receive from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$
		|	**wait** $\mathtt{p}$ | **disconnect from** $\mathtt{p}$
Computations	M, N	::=	**let** $x \Leftarrow M$ **in** N | **try** L **catch** M | **if** V **then** M **else** N | $l :: M$ | L

Syntax of Session Types

Session Actions	α, β	::=	$\mathtt{p}!\ell(A)$ | $\mathtt{p}!!\ell(A)$ | $\mathtt{p}?\ell(A)$ | $\mathtt{p}??\ell(A)$ | $\#\uparrow\mathtt{p}$
Session Types	S, T, U	::=	$\Sigma_{i\in I}(\alpha_i \,.\, S_i)$ | $\mu X.S$ | X | $\#\downarrow\mathtt{p}$ | end
Communication Actions	$\dagger$	::=	! | ?
Disconnection Actions	$\ddagger$	::=	$\#\uparrow$ | $\#\downarrow$

Figure 2.5: Syntax

describe data that has been computed. A value V can be a variable x or unit value defined by () or one of the base values. Base values consist of strings s, integers i, and booleans b (true/false).

Computations: The statement **let** $x \Leftarrow M$ **in** N first evaluates the computation M, binds its result to x, and continues to the computation N with x. As syntactic sugar, for a fresh variable x, we can write **let** $x \Leftarrow M$ **in** N as M; N. The construct **try** L **catch** M is used to carry out exception handling over a single action L. If L throws an exception, M is evaluated. $l :: M$ syntax is for labeled recursion, stating that inside the computation term M, a process can recurse to label l using the syntax **continue** l. Construct **if** V **then** M **else** N provides a conditional functionality; If value V is true, then evaluate computation M else N. Actions L consists of all the basic steps of a computation. The **return** V construct returns a value.

Concurrency and adaptation actions: The **new** u construct spawns a new actor of class u and returns its PID. The **self** construct returns the current actor's PID. An actor can replace the behaviour of itself or another actor V using **replace** V **with** κ. An actor can discover other actors following a session type S, using the **discover** S construct, which returns the PID of the discovered actor.

Session communication constructs: Syntax **connect** $\ell(V)$ **to** W **as** $\mathtt{p}$ enables an actor to connect to an actor W playing a role p, sending a message with label ℓ and a payload V. Similarly, an incoming connection can be accepted by an actor using the construct **accept from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$. This denotes accepting a connection from an actor playing role $\mathtt{p}$ along with a choice of messages. Each message consists of a label ℓ_j along with an associated payload which is bound to x_j in the followed computation M_j. As syntactic sugar, we can accept a connection with a single message as **accept** $\ell(x)$ **from** $\mathtt{p}$; M for simplicity. Once a connection has been made, an actor can communicate using the **send** and **receive** constructs. **send** $\ell(V)$ **to** $\mathtt{p}$ sends a message consisting of label ℓ and payload V to an actor playing role p. Similarly, an actor can receive a choice of messages from an actor playing role p using **receive from** $\mathtt{p}$ $\{\ell_i(x_i) \mapsto M_i\}_i$. We can also use the syntactic sugar **receive** $\ell(x)$ **from** $\mathtt{p}$; M to receive a singleton message. An actor

can disconnect from a role p using **disconnect from** p and await the disconnection of p using the construct **wait** p.

Types: Types are denoted and ranged over by A, B. These include the unit type **1**, base types - string, int, bool as well as process IDs Pid(S); S here refers to the statically-known initially defined session type of an actor, which can be found in the follows clause of the actor's definition. These types can be passed as payloads in session communications to represent the type of values involved in the exchange.

Session Actions and Session Types: Session actions, ranged over by α, β, involve the session communication actions of an actor with a participant p and message with label ℓ and type A. A session action can be one of the following; sending - p!$\ell(A)$, receiving - p?$\ell(A)$, connecting - p!!$\ell(A)$, accepting - p??$\ell(A)$ or awaiting another participant's disconnection $\#\uparrow$p.
Session types, ranged over by S, T, U, can be one of the following: a choice of action - $\Sigma_{i \in I}(\alpha_i \,.\, S_i)$, a recursive session type $\mu X.S$ binding recursion variable X in continuation session type S, a recursion variable X (appearing inside the recursive session type S), a disconnection action $\#\downarrow$p, finished session type end.
The calculus does not permit self-communication and imposes the following syntactic restrictions on session types [13]: An action choice type $\Sigma_{i \in I}(\alpha_i \,.\, S_i)$ is syntactically valid if each α_i is either a send or a connect action or it is a directed input choice, i.e., receive or accept action from a single role or a single wait action. We assume that all session types are syntactically valid for the following report.

Integrating explicit connection actions to MPST allows a participant to leave and re-join a session, or accept connections from multiple different participants. Such broadness comes at a cost since it is necessary to ensure that the same participant plays the same role throughout the session. Addressing this problem using a solution proposed by Hu & Yoshida [13], we adopt a lightweight syntactic restriction that restricts each local type to have a single accept action as its top-level construct. EnsembleS imposes this requirement as part of the safety property of Preservation and ensures that a participant's disconnection has no continuation. The behaviour of actors repeats after disconnecting and hence, a participant will be able to accept again after disconnecting and termination.

Protocols: A protocol is a set of mappings from role names to local session types. It is of the format $\{\mathsf{p}_i : S_i\}$. In the context of a program, ty(p) refers to the session type associated with the particular role, as defined by the set of protocols.

Program An EnsembleS program consists of a set of actor definitions, protocols, and an initial computation term in the 'boot' clause to be evaluated to set up initial actor communications. Thus, a program can be defined as the tuple of $(\overrightarrow{D}, \overrightarrow{P}, M)$.

2.2.2 Typing Rules

Typing rules for the provided calculus [11] is displayed in figure 2.6, 2.7 and 2.8.

Value typing, with judgement $\Gamma \vdash V{:}A$ states that under typing environment Γ, the variable V has type A (T-VAR). Typing of unit value gives unit type (T-UNIT). Similarly, typing of any of the base values - string, int, bool gives the respective type.
T-DEF: Definition typing, with judgement $\vdash D$ states that an actor definition is well-typed if its body M is typable under S and fully consumes its statically defined session type S i.e., the actor

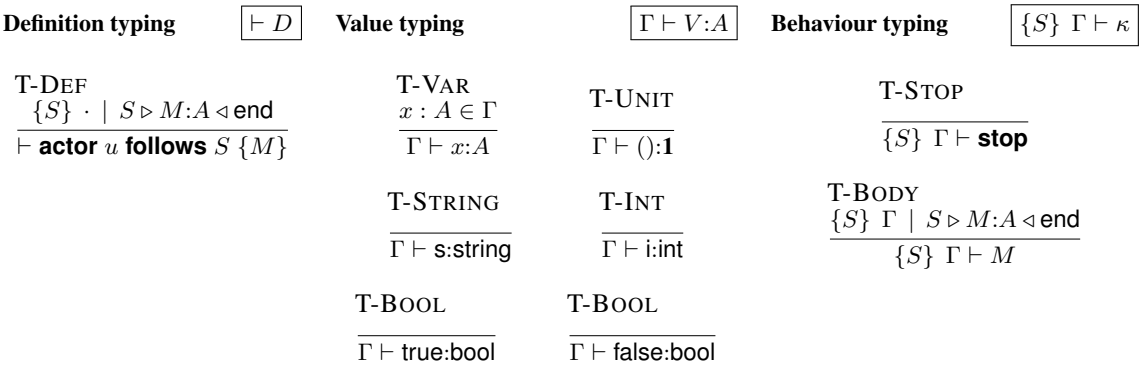


Figure 2.6: Typing rules (1)

behaviour follows and consumes the actions defined in the session type. The behaviour typing judgement $\{S\}\ \Gamma \vdash \kappa$ states the behaviour κ is well-typed under Γ for a given session type S. In this case, stop is always well-typed (T-STOP), and M is well-typed if it is typable under and fully consumes S (T-BODY).

2.2.3 Term Typing

Term typing judgement $\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'$ states that in an actor following a session type T, under the typing environment Γ and with the current session type S, term M has the type A and updates the current session type to S'. Here, S is the session precondition and the term M may perform some session communication action to update the session type to postcondition S'.

Functional Rules: Rule T-LET : Given the construct **let** $x \Leftarrow M$ **in** N, where M, under typing environment Γ, has pre-condition S and produces type A with post-condition S', and where N, under the extended typing environment $\Gamma, x : A$, has pre-condition S' and returns type B with post-condition S", the construct has an overall pre-condition S and postcondition S" with type B.
Rule T-RETURN states that type checking the construct **return** V, with pre-condition S, yields the type value of V under typing environment Γ i.e. A and post-condition S.
Rule T-REC : Given the construct $l :: M$, rule states that term M, an expression which can loop to ℓ, when evaluated using the extended typing environment $\Gamma, l : S$ gives postcondition S' and type A
Rule T-CONTINUE : This rule ensures that for the construct **continue** ℓ, the pre-condition typing environment Γ must contain the matching label ℓ. Since the behaviour recurses back to the defined label, the rule produces an arbitrary type and any post-condition dependent on the base case of the enclosing loop.
Rule T-EQUALITY and Rule T-INEQUALITY provide comparison operations; Values V, W on both side of the comparison should be of the same type A; This returns a boolean type and same post condition. According to Rule T-CONDITION, for the construct **if** V **then** M **else** N, Value V must be of type Bool and both computations M and N must have the same return type A and post-condition S'.

Actor/Adaptation Rules These rules do not update the current session type and thus, have the same pre-condition S and post-condition S.
Rule T-NEW : Rule states that spawning an actor of class u returns the PID of the declared session type. Rule T-SELF : Rule retrieves the PID for the current actor, parameterized by the

Typing rules for computations $\boxed{\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'}$

Functional Rules

T-LET
$$\frac{\{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S' \qquad \{T\}\ \Gamma, x : A\ \mid\ S' \triangleright N{:}B \triangleleft S''}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{let}\ x \Leftarrow M\ \textbf{in}\ N{:}B \triangleleft S''}$$

T-RETURN
$$\frac{\Gamma \vdash V{:}A}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{return}\ V{:}A \triangleleft S}$$

T-REC
$$\frac{\{T\}\ \Gamma, l : S\ \mid\ S \triangleright M{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright l :: M{:}A \triangleleft S'}$$

T-CONTINUE
$$\frac{}{\{T\}\ \Gamma, l : S\ \mid\ S \triangleright \textbf{continue}\ l{:}A \triangleleft S'}$$

T-EQUALITY
$$\frac{\Gamma \vdash V{:}A \qquad \Gamma \vdash W{:}A}{\{T\}\ \Gamma\ \mid\ S \triangleright V == W{:}\mathsf{Bool} \triangleleft S}$$

T-INEQUALITY
$$\frac{\Gamma \vdash V{:}A \qquad \Gamma \vdash W{:}A}{\{T\}\ \Gamma\ \mid\ S \triangleright V \neq W{:}\mathsf{Bool} \triangleleft S}$$

T-CONDITION
$$\frac{\Gamma \vdash V{:}\mathsf{Bool} \qquad \{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S' \qquad \{T\}\ \Gamma\ \mid\ S \triangleright N{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N{:}A \triangleleft S'}$$

Actor / Adaptation Rules

T-NEW
$$\frac{\mathsf{sessionType}(u) = U}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{new}\ u{:}\mathsf{Pid}(U) \triangleleft S}$$

T-SELF
$$\frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{self}{:}\mathsf{Pid}(T) \triangleleft S}$$

T-DISCOVER
$$\frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{discover}\ U{:}\mathsf{Pid}(U) \triangleleft S}$$

T-REPLACE
$$\frac{\Gamma \vdash V{:}\mathsf{Pid}(U) \qquad \{U\}\ \Gamma \vdash \kappa}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{replace}\ V\ \textbf{with}\ \kappa{:}\mathbf{1} \triangleleft S}$$

Figure 2.7: Typing rules (2)

statically-defined session type which the local actor follows. Rule T-DISCOVER : Given the term **discover** U, the construct returns a PID of type Pid(U). Rule T-REPLACE : Rule allows replacement of an actor with PID of session type U with a behaviour κ, typable under the static session type U, and returns the unit type.

Exception Handling: Rule T-RAISE denotes raising an exception. It does not return and hence, has an arbitrary return type and post-condition. Rule T-TRY types an exception handler; The action L is evaluated and if it raises an exception, then M is evaluated instead. The try and catch clauses must have the same pre-condition and post-condition to allow the action to be retried if necessary.

Session Communication Actions: Rule T-CONN: For the construct **connect** $\ell_j(V)$ **to** W **as** p_j, the pre-condition must be a choice type containing a branch $\mathsf{p}_j!!\ell_j(A_j)$. The label and payload in the construct must be compatible with the appropriate branch and value type of payload V must match type A_j in the branch. The remote actor to be connected in this session action must have a reference in W i.e. W is of type Pid(T') where T' is compatible with the type of role p_j. After this, the current session type S is changed to the matching branch's continuation S'_j and the unit type is returned. A similar procedure is followed by Rule T-SEND.
Rule T-ACCEPT: Given the current session type contains accept action(s), the rule types the term

Exception handling rules

$$\textsc{T-Raise}\ \frac{}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{raise}{:}A \triangleleft S'} \qquad \textsc{T-Try}\ \frac{\{T\}\ \Gamma\ \mid\ S \triangleright L{:}A \triangleleft S' \qquad \{T\}\ \Gamma\ \mid\ S \triangleright M{:}A \triangleleft S'}{\{T\}\ \Gamma\ \mid\ S \triangleright \textbf{try}\ L\ \textbf{catch}\ M{:}A \triangleleft S'}$$

Session communication rules

$$\textsc{T-Conn}\ \frac{\mathsf{p}_j!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j \qquad \Gamma \vdash W{:}\mathsf{Pid}(T) \qquad T = \mathsf{ty}(\mathsf{p}_j)}{\{T'\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i\,.\,S_i) \triangleright \textbf{connect}\ \ell_j(V)\ \textbf{to}\ W\ \textbf{as}\ \mathsf{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\textsc{T-Send}\ \frac{\mathsf{p}_j!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad \Gamma \vdash V{:}A_j}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\alpha_i\,.\,S_i) \triangleright \textbf{send}\ \ell_j(V)\ \textbf{to}\ \mathsf{p}_j{:}\mathbf{1} \triangleleft S'_j}$$

$$\textsc{T-Accept}\ \frac{(\{T\}\ \Gamma, x_i : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathsf{q}??\ell_i(B_i)\,.\,S_i) \triangleright \textbf{accept from}\ \mathsf{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S}$$

$$\textsc{T-Recv}\ \frac{(\{T\}\ \Gamma, x_i : B_i\ \mid\ S_i \triangleright M_i{:}A \triangleleft S)_{i\in I}}{\{T\}\ \Gamma\ \mid\ \Sigma_{i\in I}(\mathsf{q}?\ell_i(B_i)\,.\,S_i) \triangleright \textbf{receive from}\ \mathsf{q}\ \{\ell_i(x_i) \mapsto M_i\}_{i\in I}{:}A \triangleleft S} \qquad \textsc{T-Wait}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\uparrow}\mathsf{q}\,.\,S \triangleright \textbf{wait}\ \mathsf{q}{:}\mathbf{1} \triangleleft S}$$

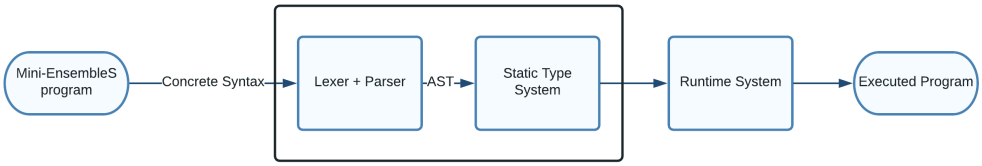
$$\textsc{T-Disconn}\ \frac{}{\{T\}\ \Gamma\ \mid\ \#{\downarrow}\mathsf{q} \triangleright \textbf{disconnect from}\ \mathsf{q}{:}\mathbf{1} \triangleleft \mathsf{end}}$$

Figure 2.8: Typing rules (3)

$\mathsf{q}??\ell_i(B_i)\,.\,S_i$ allowing an actor to accept connections from role q with messages ℓ_i, binding the payload x_i in each continuation M_i. Each of these computation terms must be typable under the the typing environment Γ extended with $x_i : A_i$, under respective session type S_i. Also, each of these branches must have the same result type A and post-condition S. Rule T-RECV is similar.
Rule T-WAIT: Rule handles the construct **wait** q where an actor is waiting for a participant q to disconnect from a session, requiring the pre-condition session type of $\#{\uparrow}\mathsf{q}$. S, returning the unit type and advancing the session type to S.
Rule T-DISCONNECT: Rule handles disconnection action from a participant q, requiring disconnection session type as pre-condition i.e. **disconnect from** q. It returns the unit type and advances the session type to end for termination.

2.3 Design

This section introduces the basic steps in the designing of a programming language and focuses on the theory of Type Systems. Figure 2.3 displays the step-by-step breakdown of the design process adopted for Mini-EnsembleS programs. The program source code is fed into the pipeline; the first stage is the parser, which translates the input code into an Abstract Syntax Tree, followed by the type checker and the runtime systems to execute this intermediate output. The implementation of this project involves the boxes stages in figure 2.3 i.e., Parser & Type Checker.



2.3.1 Parser

The Parser's task is to turn a list of tokens into a tree of nodes adding structure to it [7]. It takes the text of our programs and interprets which commands the code expresses. It recognizes statements, expressions and creates internal data structures to represent them - an Abstract Syntax Tree or AST. AST provides a method to represent the input code as a higher-level data structure, eliminating information that is not crucial for processing the commands in the following stages. The next stages in the pipeline will not refer back to the original source code, so the parser must extract all the required information. It is beneficial to encapsulate this labelling logic inside the parser at the initial stages of the pipeline so as to not worry about these rules at further stages as well as the flexibility this provides to change the syntax at this base level itself.

2.3.2 Type System

The purpose of a type system is to reduce the bugs in computer systems [3], by creating interfaces between different segments and then checking that these components are connected meaningfully. A type system is a logical framework that associates a type to different structures of a program, such as variables, expressions and functions and, by examining the exchange of these values, prevents type errors from occurring. A program that passes a type checker is guaranteed to meet a set of type safety characteristics for all potential inputs.

The expression $\mathsf{e} : \tau$ indicates that the term e is of the type τ. For example, 1, 2+3, 4 * 5 are all separate terms each of type int for integers; true, false are terms of type bool. A typical judgment has the form: $\Gamma \vdash \xi$, where ξ is an assertion and Γ is a static typing environment under which typing takes place [3]. Often, an environment is a list of pairs e:τ i.e., distinct variables and their types.
A type judgment is of the form $\Gamma \vdash M{:}A$, which asserts that a term M has a type A under the typing environment Γ. For example, for empty environment ϕ, $\phi \vdash true{:}Bool$, which implies, true is of type Bool. $\phi, x : Nat \vdash x + 1{:}Nat$ implies type of x+1 is Nat if x is of type Nat.

Type rules assert the validity of certain judgments based on other judgments that are already known to be valid [3]. Each type rule is stated as a series of premise judgments $\Gamma_i \vdash \xi_i$ above a horizontal

$$\frac{\text{(RULE NAME)}}{\Gamma_1 \vdash \xi_1 \quad \ldots \quad \Gamma_n \vdash \xi_n}{\Gamma \vdash \xi}$$

Figure 2.9: General form of a type rule

line, followed by a single conclusion judgment $\Gamma \vdash \xi$. The conclusion must hold when all of the premises are satisfied. The number of premise judgments may be zero as well. A type system is a set of type rules.

Chapter 3

Implementation

This chapter describes the design and implementation of a parser and typechecker for Mini-EnsembleS. We designed a smaller, clean language to present the ideas of the calculus in an articulate manner. The concrete syntax of this core language is presented in section § A.1. The first stage of the implementation begins with parsing the input (a code using concrete syntax) into an abstract syntax tree, which is then fed into a type checker. The type checker catches the type errors as expressed by the typing rules in section § 2.2.2. The functional programming language Haskell is used because of its support of algebraic data types and ability to express the formal calculus in a pure and mathematical method, along with sophisticated tools for parsing and type checking.

3.1 Abstract Syntax Tree

This sections describes the concepts related to abstract syntax and its implementation in our project. The abstract syntax tree attempts to reflect just the most important features of our calculus's structure while omitting non-essential, representation-dependent characteristics of the concrete syntax. For example, parentheses convey structural information; there is no need for parenthesis in the abstract syntax since this structural information may be conveyed directly in the abstract syntax. We can construct abstract syntax trees with ease in Haskell using algebraic data types. By employing pattern matching, we can define concise functions that traverse the AST. Following are few examples of ASTs.

```haskell
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Val Int

-- ( 1 + 2 ) * 3
-- Mul (Add (Val 1) (Val 2)) (Val 3))
```

```haskell
data BinaryTree a = Leaf
  | Node a (BinaryTree a)(BinaryTree a)
--          [1]
--      [2]     [3]
--  Node 1 (Node 2 Leaf Leaf) (Node 3
    Leaf Leaf)
```

The abstract syntax tree implemented for the Mini-Ensemble language is described in the appendix section § A.2.

3.2 Parser

A parser is a part of a compiler or interpreter that divides the input data, which might be a sequence of tokens, interactive commands, or computer instructions into smaller chunks that can be used by other programming components in the pipeline. Megaparsec [16], a Haskell monadic parser combinator library, is used in our implementation. For a long period, Parsec was the standard Haskell parsing library [14]. It was built from the ground up to be an industrial-strength parser library. It's easy to use, safe, extensive libraries and documentation, and focuses on the quality of error messages. Megaparsec is derived from this library; it strikes a good blend of performance, flexibility, error messaging, and readability.

3.2.1 Parsing environment

`ParsecT`, defined as `ParsecT e s m a` is the central data type in Megaparsec [14]. Here, `e` is the type of custom error messages, `s` is the input stream, `m` is the inner monad of ParsecT transformer, `a` is the resultant monadic value. Using the non-transform version of ParsecT, Parsec, we define our central parsing component - `Parser` as: `type Parser = Parsec Void Text`. Defined type deals with Text input, without integrating any custom error component.
The implementation of all the high-level parsers involve breaking down the input text and combining them by using appropriate parsers in succession. Implementing the combinators in table 3.1, the parser constructs the Abstract Syntax Tree, as defined in figure A.2, from the source code.

many	*many p* applies the parser p zero or more times and returns a list of the values returned by p.
between	*between open close p* parses open, followed by p and close. Returns the value returned by p.
choice (< | >)	applies the parsers in the provided list in order, until one of them succeeds. Returns the value of the succeeding parser.
sepBy	*sepBy p sep* parses zero or more occurrences of p, separated by sep. Returns a list of values returned by p.
try	*try p* behaves like the parser p, except that it backtracks the parser state when p fails.
string	*string str* only matches the provided string str and returns it.
string'	Same as **string**, but case insensitive.

Table 3.1: Commonly used parser combinators

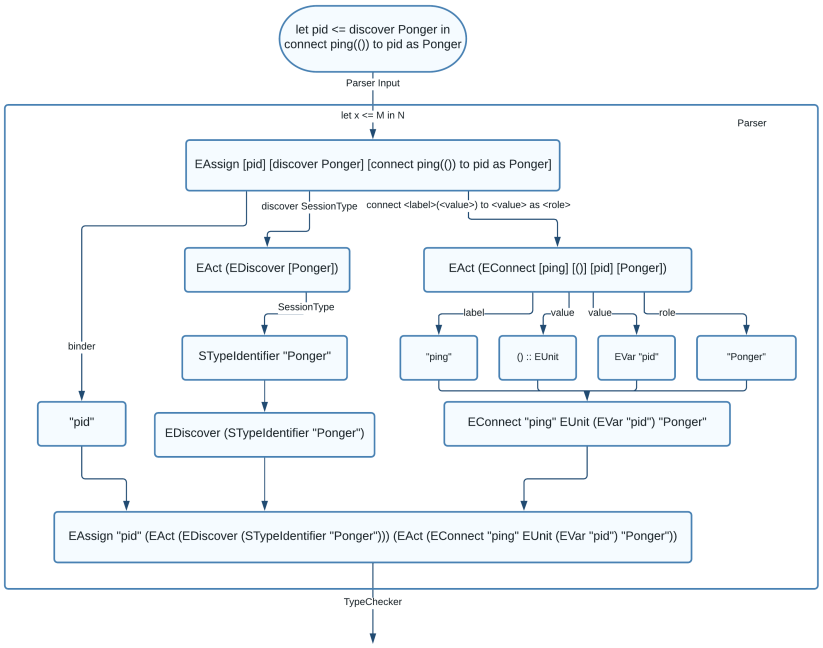
For example, the combinators `between` and `many` are used to define a higher-level parser - `identifier`, that extracts a value from square brackets.

```
> squareBrackets = between (symbol "[") (symbol "]")
> identifier = squareBrackets (many alphaNumChar)
> parseTest identifier "[value123]"
 "value123"
```

3.2.2 Running the parser

The implemented library, Megaparsec, uses monadic parser combinators which collapse the input into smaller chunks of data for independent processing. The output of this process is an Abstract Syntax Tree; We pipeline this output to the type checker.

Figure 3.2.2 shows the flow of how parsing combinators work. For this example, the code input for the parser is `let pid <= discover Ponger in connect ping(()) to pid as Ponger`. Using pattern matching, the input is matched with the construct for the computation **let** $x \Leftarrow M$ **in** N. Following this, each of x, M and N are parsed separately. x is parsed as a `Binder` (String); M is the computation **discover** $Ponger$ which is parsed as the action `EAct (EDiscover (STypeIdentifier "Ponger"));` N is parsed as the connect action `EAct (EConnect "ping" EUnit (EVar "pid") "Ponger")`. All of these are combined to give us the displayed output in the form of a high-level abstract tree.



3.3 Type Checker

The type checker plays a central role in the implementation of this project. The typing rules discussed in § 2.2.2 are implemented in this section; the developed code checks the parsed input for type errors at compile-time and validates the system's correctness. To structure error handling, we use the monad transformer defined in Control.Monad.Except [17] and Data.Either type [9].

3.3.1 The Either type

The Either type, defined in the module Data.Either, represents values with two possibilities. Value of `data Either a b` will be either `Left a` or `Right b`. While using it for error handling, the Left constructor is used to represent error values and the Right constructor holds correct/success values. For example, we use pattern matching to detect and throw an error in the function below.

```
safeDivision :: Float -> Float -> Either String Float
safeDivision x 0 = Left "DivisionByZero"
safeDivision x y = Right (x / y)
```

The function safeDivision takes two floats as input and returns Either String Float i.e. either a String type or a Float type, depending on the processing. In case the second float input is a zero, we return the String "DivisionByZero" to indicate the error; in other cases, we return the quotient of the division operation (x/y). In this manner, we can use the Either type to provide a choice between returning values in a successful run or throwing an error.

Haskell provides mechanisms to build computations from sequences of functions that may throw an error. In case of a failure, the cause and location of the error can be recorded, wrapped, and passed to the error-handling mechanism. The strategy of bypassing the bound functions where the exception occurs to the mechanism in the program where it is handled enables us to handle errors in a convenient and abstract fashion. An example of error-handling is shown below.

```haskell
-- Type to represent mobile number validation
type MobileMonad = Either String
-- Function returns 100 if valid extension code and length
-- Failure is Left String
-- Success is Right Int
calNumber :: String -> MobileMonad Int
calNumber [] = throwError "Empty input"
calNumber s | ext /= "+44" = throwError "Invalid Country code"
                  | len /= 13 = throwError "Invalid number"
                  | otherwise = return 100
  where ext = take 3 s
```

The above example displays how a function can deal with errors using the mechanism throwError, which initiates the exception handling. Different erroneous inputs are handled with the Left String return value; Successful run returns type Right Int / Right 100. We can use a custom error data type to handle exceptions conveniently.

3.3.2 The custom Error data type

We use a custom error type to catch type errors as defined in the typing rules of our calculus.

```haskell
data Error = UnboundVariable String
    | UnboundRecursionVariable String
    | ActorNotDefined String
    | IncompatibleTypes Type Type
    | SessionMismatch SessionType SessionType
    | BranchError Role Label
    | BranchRoleError Role
    | SessionConsumptionError SessionType
    | UndefinedProtocol Role
    | RoleMismatch Role
    | UndefinedAlias SessionType
    | ExpectedPID Type
instance Show Error where
    show (UnboundVariable val) = "Unbounded variable: " ++ (show val)
```

The error **UnboundVariable** is thrown when a variable cannot be found in the typing environment while type-checking a value. Error **ActorNotDefined** is thrown when an undefined actor variable name is used in syntax constructs. **IncompatibleTypes**, **SessionMismatch** exceptions occur when comparing types or sessions for equality. If an expected session action is not found in the current session type, a **BranchError** or **BranchRoleError** (in case of wait <role>) exception is thrown. If a session is not consumed fully by an actor's computation, **SessionConsumptionError** occurs. **UndefinedProtocol** error indicates a missing protocol entry for a particular role. **RoleMismatch** indicates an inconsistent role entry in the session type and computation term. **UndefinedAlias** exception occurs if a references session type has no type alias defined. **ExpectedPID** error is thrown when an input value is not of the required type PID(Session).
Using this custom set of exception handlers, the type checker validates the input, according to the mentioned typing rules.

3.3.3 Typing Environment and State

The typing environment i.e., Γ described in the typing rules is implemented using two lists in the type checker, namely, TypeEnv and RecEnv. These are defined as follows:

```
type TypeEnv = [(String, Type)]
type RecEnv = [(Label, SessionType)]
```

`TypeEnv` stores entries in the form of tuple; each entry is a variable name with it's corresponding type. During value typing i.e. $\Gamma \vdash x{:}A$, we lookup the value x in this typing environment and compare this with the expected type A. This typing environment is extended during variable bindings and term continuations.
`RecEnv` is used specifically to enable recursion using labels in computation terms. If a labelled recursion of the form $l :: M$ is encountered with the current session type denoted as **S**, the recursion environment RecEnv is extended with an entry **(l, S)** to enable the process to recurse back to label l using the construct **continue** l.
Term typing $\{T\}\ \Gamma \mid S \triangleright M{:}A \triangleleft S'$ requires the type checker to keep track of the initial session type T, the typing environment Γ, current session type S, as well as list of actor definitions, protocols and defined session type aliases for reference. We encapsulate this data chuck into a record called State and this data type is passed around and updated as required. We define it as:

```
data State = State {
    actorDefs :: [ActorDef]     -- List of actor definitions
  , protocols :: [Protocol]     -- List of protocols
  , typeAliases :: [TypeAlias]  -- List of type aliases
  , typeEnv :: TypeEnv          -- TypeEnv for value checking
  , recEnv :: RecEnv            -- RecEnv for labelled recursion
  , followST :: SessionType     -- Initially defined session type
  , session :: SessionType      -- Current Session Type
}
```

3.3.4 Equi-recursive Session types

Typing theory for EnsembleS identifies equi-recursive view of session types i.e. recursive session types with their unfolding. This implies, $\mu X.S = S\{\mu X.S/X\}$ where, recursion is assumed to be guarded. To implement this, we resolve a session type of the form **rec X.S** to **S . rec X.S**.
As an illustration, the following session types are equi-recursive:

```
rec keepPing . ping !! pong(String) . keepPing
ping !! pong(String) . rec keepPing . ping !! pong(String) . keepPing
```

Our implementation considers 1-unrolling of session types when checking for equivalence using the `resolveSessionType` function. This can be easily extended to adapt to multiple unrollings.

3.3.5 Structure of Type System

We make use of the Either type and a custom Error data type to process exception handling; Left constructor hold one of the error values and Right constructor holds the success return type. We define a type:

```
type TypeCheck = Either Error
```

A Mini-EnsembleS program can be represented by a 4-tuple set - $(\vec{TA}, \vec{D}, \vec{P}, M)$, where $\vec{TA}$ is the set of all defined session type aliases, $\vec{D}$ is the set of all actor definitions, $\vec{P}$ is the list of all protocols and M is the initial computation term. According to the typing rules, the top-down approach to type-checking this program is as follows:

```
checkProgram :: Program -> TypeCheck ()
checkActorDef :: State -> ActorDef -> TypeCheck ()
checkComputation :: State -> Computation -> TypeCheck (Type, SessionType)
checkValue :: TypeEnv -> EValue -> TypeCheck Type
```

`checkProgram` is a function that takes in a type Program, as defined in the AST, performs type-checking and throws errors if any, or returns (). Similarly, `checkActorDef` function performs type checking on the input actor definition; it accepts a State type and returns () on success. Term typing rules dictate type-checking on a Computation should return a type and session type. `checkComputation` is called on a Computation to implement the various term typing rules by use of pattern matching. Lastly, value typing is carried out using `checkValue` function, which accepts a typing environment TypeEnv and a value constructor EValue and returns its Type.

3.3.6 Executing Type-Checker

The parsed output, in the form of an Abstract Syntax Tree, is processed for type checking. This input to the type checker is of the type Program and the function `checkProgram` is the entry point in the code. This function in turn calls the `checkActorDef` on all the defined local actor definitions as well as a `checkComputation` call on the initial term M. `checkActorDef` uses the input ActorDefinition type, which contains the local actor's session type and behaviour, and ensures that the body is typable under and fully consumes the session type.

Value typing works in a similar manner; figure 3.3.6 demonstrates how pattern matching is used to type check a value according to the value typing rules.

$$\frac{}{\Gamma \vdash ():\mathbf{1}}\ \text{T-UNIT} \qquad \frac{}{\Gamma \vdash \text{s:string}}\ \text{T-STRING} \qquad \frac{}{\Gamma \vdash \text{i:int}}\ \text{T-INT} \qquad \frac{}{\Gamma \vdash \text{true:bool}}\ \text{T-BOOL} \qquad \frac{}{\Gamma \vdash \text{false:bool}}\ \text{T-BOOL} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x{:}A}\ \text{T-VAR}$$

```
-- VALUE TYPING
checkValue :: TypeEnv -> EValue -> TypeCheck Type
checkValue _ EUnit       = return Unit         -- T-UNIT
checkValue _ (EString _) = return TString      -- T-STRING
checkValue _ (EBool _)   = return TBool        -- T-BOOL
checkValue _ (EInt _)    = return TInt         -- T-INT
checkValue typeEnv (EVar string) = do          -- T-VAR
  let returnType = Map.lookup string $ Map.fromList typeEnv in
    case returnType of
      Just t -> return t
      Nothing -> throwError (UnboundVariable string)
```

To illustrate how value typing is implemented, a few examples are enlisted below. In cases where input is a base value like String, Bool, Int, or Unit, checkValue returns the corresponding base type. If a variable value is given as input, the typing environment is looked up and if the entry exists, its type is returned else an UnboundVariable exception is thrown.

```
> checkValue [] EUnit
Right Unit
> checkValue [("x", TString)] (EString "abc")
Right TString
> checkValue [("x", TInt)] (EVar "x")
Right TInt
> checkValue [("x", TInt)] (EVar "y")
Left Unbounded variable "y"
```

Consider the implementation of rules T-LET, T-DISCOVER and T-CONNECT:

$$\frac{\{T\}\ \Gamma \mid S \triangleright M{:}A \triangleleft S' \qquad \{T\}\ \Gamma, x : A \mid S' \triangleright N{:}B \triangleleft S''}{\{T\}\ \Gamma \mid S \triangleright \textbf{let}\ x \Leftarrow M\ \textbf{in}\ N{:}B \triangleleft S''}\ \text{T-LET}$$

$$\frac{}{\{T\}\ \Gamma \mid S \triangleright \textbf{discover}\ U{:}\mathsf{Pid}(U) \triangleleft S}\ \text{T-DISCOVER}$$

$$\frac{(\mathbf{1})\mathsf{p}_j!!\ell_j(A_j) \in \{\alpha_i\}_{i\in I} \qquad (\mathbf{2})\Gamma \vdash V{:}A_j \qquad (\mathbf{3})\Gamma \vdash W{:}\mathsf{Pid}(T) \qquad (\mathbf{4})T = \mathsf{ty}(\mathsf{p}_j)}{\{T'\}\ \Gamma \mid \Sigma_{i\in I}(\alpha_i \,.\, S_i) \triangleright \textbf{connect}\ \ell_j(V)\ \textbf{to}\ W\ \textbf{as}\ \mathsf{p}_j{:}\mathbf{1} \triangleleft S'_j}\ \text{T-CONN}$$

The code snippet describes the mechanism followed for the type checking of rules described above. The **EAssign** constructor binds the type return by computation `c1` in `binder` and evaluates the computation `c2` with the extended typing environment containing (binder,ty) entry. Type checking **EDiscover** action return the type (EPid s, current_session). For **Econnect** action, the `role` and `label` are used to search for a connect action in line 12, which returns the corresponding type - `typeA` and updated session type - `session'` from the matching session type (Premise **(1)**). Values `valueV` and `valueW` are type checked. `typeW` expected to be of type $\mathsf{Pid}(T)$, is unwrapped to give

```
1  .
2  checkComputation state (EAssign binder c1 c2) = do
3    (ty, session') <- checkComputation state c1
4    (ty', session'') <- checkComputation state {typeEnv=(binder,ty):(typeEnv
       state), session=session'} c2
5    return (ty', session'')
6  .
7  .
8  checkComputation state (EAct (EDiscover s)) = getAlias (typeAliases state) s
       >>= \s -> return (EPid s, (session state))
9  .
10 .
11 checkComputation state (EAct (EConnect label valueV valueW role)) = do
12   (typeA, session') <- getConnectAction (session state) role label     -- (1)
13   typeV <- checkValue (typeEnv state) valueV
14   typeW <- checkValue (typeEnv state) valueW
15   sessionW <- unwrapPID typeW
16   sessionType' <- getSessionTypeOfRole (protocols state) role
17
18   compareTypes (typeAliases state) typeV typeA                          -- (2)
19   compareSessions (typeAliases state) sessionW sessionType'          -- (3),(4)
20   return (Unit, session')
21 .
```

Figure 3.1: Typechecking source code for T-LET, T-DISCOVER & T-CONN

its session type `sessionW` (line 15). Session type `sessionType'` of `role` is obtained from the protocol mappings (line 16). *typeV* is compared with *typeA* at line 18 to ensure type of payload is same in the defined session type and the actor behaviour (Premise **(2)**). *sessionW* is compared with the session type `sessionType'` (line 19) to ensure safe connection action (Premise **(3,4)**). The complete sequence has a return type of (Unit, *session'*) (line 20) if program is well-typed.

All the rules have been similarly implemented to ensure that a local actor conforms to its defined session type. Haskell's features, based on Functional programming paradigm, such as recursion, pattern matching, lambda expressions, and use of monads enable us to implement all the typing rules, in a purely functional manner. These functions will only depend on their input arguments, regardless of other states, and perform the necessary evaluation in an abstracted environment; Further processing and other side effects can be handled completely independent of this implementation and thus, provides a suitable foundation in the process of type-checking and executing a Mini-EnsembleS program.

Chapter 4

Evaluation

In this chapter, we illustrate the implementation of our type-checker using three instances - Ping-Pong scheme, an OnlineStore example and non-trivial DNS client-server interaction adapted from EnsembleS [11]. We evaluate these examples on the ability of our system to express these schemes with more ease and without implementation artifacts, as seen in previously designed EnsembleS programs, as well as the type checker's capacity to detect and prevent potential type errors that threaten communication safety. All examples illustrate how session types and explicit communication actions are implemented in different scenarios.

We start this section with an example, as discussed in parsing example 3.2.2. The relevant source code for this example is listed in figure 3.3.6. To execute this code, let us define few session types and custom states, covering some of the potential erroneous inputs. `action` contains the required parsed action to be typechecked and we use the `checkComputation` function to typecheck the action. Session type `ses1` contains a connect action with a string type payload (incompatible with required value type in defined action); session `ses2` contains the matching connect action for our computation. The defined states, s1-s5 each have some missing or incompatible entity, resolved sequentially till state s6 which gives the expected output.

```
action = EAssign "pid" (EAct (EDiscover (STypeIdentifier "Ponger")))
        (EAct (EConnect "ping" EUnit (EVar "pid") "Ponger"))
ses1 = SAction [(SConnect "Ponger" "ping" TString, SEnd )]
ses2 = SAction [(SConnect "Ponger" "ping" Unit,
            SAction[(SWait "Ponger", SEnd)] )]
s1 = State {typeAliases = [], actorDefs = [], protocols = [],
        typeEnv = [], recEnv = [], followST = SEnd, session = SEnd }
s2 = s1 {typeAliases = [(SessionTypeAlias (STypeIdentifier "Ponger")
            (SAction[(SWait "Pinger", SEnd)]) )]}
s3 = s2 {session = ses1}
s4 = s3 {protocols = [Protocol "Ponger" SEnd]}
s5 = s4 {session = ses2}
s6 = s5 {protocols = [Protocol "Ponger" (SAction [(SWait "Pinger", SEnd )])]}

> checkComputation s1 action
Left Session "Ponger" has no defined type alias
```

```
> checkComputation s2 action
Left "Connect" action for role "Ponger" and label "ping" does not exist

> checkComputation s3 action
Left Role "Ponger" has no defined protocol

> checkComputation s4 action
Left Types: Unit and TString are not compatible

> checkComputation s5 action
Left Mismatch of SessionTypes: SAction [(SWait "Pinger",SEnd)] and SEnd

> checkComputation s6 action
Right (Unit,SAction [(SWait "Ponger",SEnd)])
```

State s1 has no typeAliases defined, hence, function getAlias at line 8 in source code (3.3.6) throws a **UndefinedAlias** exception when it encounters the construct *EDiscover(STypeIdentifier "Ponger")*. State s2 is defined with a typeAlias, but does not contain the required connect action in its session type; it throws **BranchError** in the function getConnectAction at line 12, with the error message as shown. State s3 is defined using session type ses1. However, this state does not have a protocol entry mapping role "Ponger" to its session type hence throws an **ActorNotDefined** exception in function at line 15. State s4 has a protocol mapping but a **TypeMismatch** error is thrown as the action tries to send a EUnit value of type Unit and ses1 expects a type TString (line 14). Computation with state s5 throws a **SessionMismatch** error at line 15 when comparing role "Ponger"'s session type and session type value in valueW. When all the premise judgements are satisfied, the computation with state s6 gives a successful type check output with a Unit type and the updated session type.
All the discussed typing rules follow a similar structure, where components of each rule are type-checked, following a pattern to detect errors or indicate a well-typed program.

4.1 Ping-Pong

The Ping-Pong scheme can be observed in various field of software engineering such as computer networking and databases. It is characterized by an exchange between two entities - **Pinger** and **Ponger**. In networking, this scheme is used as a tool to test the reachability of a host; The protocol initiates with the Pinger entity requesting a connection to a destination host Ponger by sending a data packet and the host sends a response back to the source to conclude the protocol.

The program written in 4.1 displays the Mini-Ensemble code for a ping-pong scheme.
Lines 1,2 are the type aliases defined for this scheme. According to `PongerSession` defined at line 1, the actor accepts a connection from `Pinger` with a message label "ping" and Unit type. Following this, actor sends message "pong" with Unit type to role `Pinger` and disconnects from it. This behaviour can be observed in the actor definition of PongerActor (lines 11-17)
Complementing this, in `PingerSession` at line 2, the actor connects to role Ponger with message "ping" and payload type Unit. It receives a response message from `Ponger`, awaits its disconnection

```
1  type PongerSession = Pinger ?? ping(Unit) . Pinger ! pong(Unit) . ##Pinger
2  type PingerSession = Ponger !! ping(Unit) . Ponger ? pong(Unit) . #Ponger .
       end
3  actor PingerActor follows (PingerSession) {
4      let pid <= discover PongerSession in
5      connect ping(()) to pid as Ponger;
6      receive from Ponger {
7          pong(x) -> wait Ponger
8      }
9  }
10 actor PongerActor follows (PongerSession) {
11     accept from Pinger {
12         ping(x) ->
13             send pong(()) to Pinger;
14             disconnect from Pinger
15     }
16 }
17 Pinger : PingerSession
18 Ponger : PongerSession
19 boot { new PingerActor; new PongerActor }
```

Figure 4.1: EnsembleS Ping-Pong Scheme

and terminates session. Lines 3-10 depict this behaviour; `PingerActor` connects to `PongerActor` by discovering its pid and follows the communication manner as described in the scheme. `Pinger` actor explicitly connects to `Ponger` using the connect construct at line 5. Lines 17,18 are protocol definitions mapping role "Pinger" to "PingerSession" and "Ponger" to "PongerSession". Line 19 is the boot clause to initiate the actor communication.
The program in 4.1 represents the described scheme and when fed into the parser and subsequently the type checker does not throw any type errors. If we make changes to the input code in various constructs, we get the following errors:

```
> ./typecheck ping_input
 "No type errors"
-- Change line 1
-- Pinger ?? ping(Unit) -> Pinger ? ping(Unit)
> ./typecheck ping_input
 "Accept" action for role "Pinger" and label "ping" does not exist
-- Change line 7
-- wait Ponger -> return "value"
> ./typecheck ping_input
 Incomplete consumption of session: "PingerSession"
-- Change line 13
-- send pong(()) to Pinger -> send pong(100) to Pinger
> ./typecheck ping_input
 Types: TInt and Unit are not compatible
```

Similarly, different type errors can be detected and prevented at compile-time to ensure type safety in this scenario.

4.2 Online Store

The online store example describes a shopping service involving a Customer, a Store, and a Courier service. In this protocol, a customer can browse and ask for quotes for items repeatedly from the store. Alternatively, the customer can also ask to deliver an item in which case, it sends its address to the store. The store connects with a courier service and returns the reference number provided by the courier to the customer. This can be observed in the provided session types in figure 4.2

```
1  type Customer = Store !! login(String) . rec browse .
2       (Store ! item(String) . Store ? price(Int) . browse)
3     +  (Store ! buy(String) . Store ! address(String)
4       . Store ? ref(Int). #Store. end)
5     +  (Store ! quit(Unit) . #Store . end)
6  type Store = Customer ?? login(String) . rec browse .
7       (Customer ? item(String) . Customer ! price(Int) . browse)
8     + (Customer ? buy(String) . Customer ? address(String)
9       . Courier !! deliver(String) . Courier ? ref(Int)
10      . #Courier . Customer ! ref(Int) . ##Customer)
11    + (Customer ? quit(Unit) . ##Customer)
12 type Courier = Store ?? deliver(String) . Store ! ref(Int) . ##Store
```

Figure 4.2: Session types for OnlineStore

Each connection in the described protocol is established explicitly; connection action from Customer to Store (line 1) or from Store to Courier (line 9) are explicit and enable a protocol to involve actors only when required. Connection to Courier actor is only initiated when required by the Store to deliver an item (lines 8-10) and not in other branch actions.

In the program described in 4.2, Customer1 behaviour follows the first branch of session type described in 4.2; the actor asks for a quote from Store, receives a price, and continues to browse (line 1-7). Customer2 proceeds to buy an item from the store by sending its address and subsequently receiving a reference number for the delivery (line 8-15). A CourierActor simply accepts a connection from Store for a delivery, sends a reference number back to the store, and closes the connection (line 16-19). A store actor accepts a connection from a customer and follows the behaviour in the labeled recursion "browse". Depending on the received message from the Customer, the store either sends a quote for an item (line 24-26), follows the protocol for purchasing an item (line 27-34), or terminates the connection with the customer (line 35).
All interactions in this program have been initiated via the discovery process followed by explicit connection actions. This facilitates an adaptable system; the customer can connect to a store based on its vicinity or other requirements and similarly a store can choose which courier service to contact depending on delivery address or the item's specifications.

The source code displayed in 4.2 is well-typed and throws no type errors. To demonstrate our typechecker's implementation, we introduce few errors in the input code. According to the calculus, the construct "**connect** $label(valueV)$ **to** $valueW$ **as** $role$" requires `valueW` to be of type `Pid(S)`, where S is the mapped session type of `role`. Changing the type of this `valueW` will compromise the T-CONN typing rule (refer figure 2.8). Similarly, changing the session 'Store' in the discovery process in line 2 to other session types should be detected as a communication error by our typechecker; let us change the type and see the output from the typechecker.

```
actor Customer1 follows (Customer) {
    let pid <= discover Store in
    connect login("credentials") to pid
    as Store;
    browse ::
      send item("COVID kit") to Store;
      receive price(x) from Store;
      continue browse }
actor Customer2 follows (Customer) {
    let pid <= discover Store in
    connect login("credentials") to pid
    as Store;
    browse ::
      send buy("COVID kit") to Store;
      send address("Kelvinhaugh Street")
    to Store;
      receive ref(x) from Store;
      wait Store }
actor CourierActor follows (Courier) {
    accept deliver(addr) from Store;
    send ref(100) to Store;
    disconnect from Store }
```

```
actor StoreActor follows (Store) {
  accept login(credentials) from
    Customer;
  browse ::
    receive from Customer {
      item(itemName) ->
        send price(20) to Customer;
        continue browse
      buy(item) ->
        receive address(addr) from
  Customer;
        let pid <= discover Courier in
        connect deliver(addr) to pid as
  Courier;
        receive ref(x) from Courier;
        wait Courier;
        send ref(x) to Customer;
        disconnect from Customer
      quit(x) -> disconnect from
  Customer
    }
}
```

Figure 4.3: Mini-EnsembleS OnlineStore

```
> ./typecheck dns_input
 "No type errors"

-- Change in line 3
-- pid -> "stringValue"
> ./typecheck dns_input
 "Type error for type TString. Expected PID"

-- Change in line 1
-- discover Store -> discover Customer
> ./typecheck dns_input
 "Mismatch of SessionTypes: 'Customer' and 'Store'"

-- discover Store -> discover GlaGiftShop
> ./typecheck dns_input
 "Session 'GlaGiftShop' has no defined type alias"
```

From the above examples, we can conclude that the developed typechecker can validate applications by ensuring the local types conform to their defined session types.

```
1  type ClientSession = Root !! RootRequest(String) . (Root ? InvalidTLD(String)
       . #Root . end) + (Root ? TLDResponse(Pid(ZoneSession)) . #Root . ZS !!
     ResolutionRequest(String) . rec lookup . (ZS ? PartialResolution(Pid(
     ZoneSession)) . #ZS . ZS !! ResolutionRequest(String) . lookup) +  (ZS ?
     InvalidDomain(String) . #ZS . end) + (ZS ? ResolutionComplete(String) . #
     ZS . end))
2
3  actor Client follows (ClientSession) {
4    let rootServer <= discover RootSession in
5    connect RootRequest("uk") to rootServer as Root;
6    receive from Root {
7      TLDResponse(zoneServerAddr) ->
8        wait Root;
9        connect ResolutionRequest("www.gla.ac.uk") to zoneServerAddr as ZS;
10       lookup ::
11         receive from ZS {
12           PartialResolution(pidZS) ->
13             wait ZS;
14             connect ResolutionRequest("www.gla.ac.uk") to pidZS as ZS;
15             continue lookup
16           InvalidDomain(invalidString) ->
17             wait ZS
18           ResolutionComplete(ipAddress) ->
19             wait ZS
20         }
21     InvalidTLD(invalidString) ->
22       wait Root
23   }
24 }
```

Figure 4.4: DNS Client

4.3 DNS server

Domain Name System (DNS) is a hierarchical and decentralized system used by computers to identify other devices, services and resources on the internet. It maps domain names, such as "www.gla.ac.uk", to its corresponding Internet Protocol (IP) address, which is used by the computers to locate and communicate resources, using network protocols.
The client initiates a DNS lookup by sending the domain name to one of the root servers. The root server either refers to the zone server responsible for that particular domain or rejects the request if considered invalid. A zone server may be able to resolve the domain name and sends the IP address to the client or it may refer the request to another zone server. This recursive lookup continues until the domain name is resolved or if it cannot be found.
Figure 4.4 shows the defined session type and the definition of a Client actor that follows the session. The client uses an explicit discovery action to connect with the root server (line 4) and zone servers (line 9) using their associated session types; this makes our system adaptable and able to operate in a dynamic network. If the root server informs that the request is invalid (line 21), the client awaits disconnection from root server and the session terminates. If the request is valid, root server

```
1  type Iclient is interface(
2   out{RootServer,string} RootServer_stringOut,
3   in {RootServer,string} RootServer_stringIn,
4   out{ZoneServer,string} ZoneServer_stringOut,
5   in {ZoneServer,string} ZoneServer_stringIn,
6   in {ZoneServer,choice_enum} ZoneServer_choiceIn,
7   in {RootServer,choice_enum} RootServer_choiceIn)
8
9  type choice_enum is
10  enum(TLDResponse,PartialResolution,
11  InvalidDomain,ResolutionComplete,
12  InvalidTLD)
13
14 query find_name(string n){ $name == n; }
15
16 actor c presents Iclient
17  follows Client {
18   dom_name = "nii.ac.jp";
19   constructor() { }
20   behaviour{
21    rootQuery = find_name("jp");
22    // Find Root Server
23    root_s =
24    discover(IServer, RootServer, rootQuery);
25    // search until root_s non-empty
26    link me with root_s[0];
27    send domain_name on RootServer_stringOut;
28    receive c_msg from RootServer_choiceIn;
29    switch(c_msg){
30     case TLDResponse:
31      receive ZoneServerAddr_msg
32          from RootServer_stringIn;
33        unlink RootServer;
34   while(true) Lookup : {
35   // Find ZoneServer
36   zone_s = discover(IServer, ZoneServer, find_name(
         ZoneServerAddr_msg));
37   link me with zone_s[0];
38  // Ask ZoneServer
39   send dom_name on ZoneServer_stringOut;
40   receive c_msg2 from ZoneServer_choiceIn;
41   switch(c_msg2){
42    case PartialResolution:
43     receive str_msg from ZoneServer_stringIn;
44     ZoneServerAddr_msg := str_msg;
45     unlink ZoneServer;
46     continue Lookup;
47    case InvalidDomain:
48     receive str_msg from ZoneServer_stringIn;
49     unlink ZoneServer;
50     break;
51    case ResolutionComplete:
52     receive str_msg from ZoneServer_stringIn;
53     unlink ZoneServer;
54     break Lookup;
55    }
56    // keep looking
57   }
58   case InvalidTLD:
59    receive str_msg from RootServer_stringIn;
60    unlink RootServer;
61  } } }
```

Figure 4.5: EnsembleS DNS Client

sends the PID of respective zone server (line 7) which is then used to establish communication. The client recursively queries zone servers (lines 10-15) in the case of partial resolutions of the request until either a error in resolution occurs (lines 16-17) or the resolution is complete (lines 18-19). We can observe that the behaviour of Client actor follows the defined session type ClientSession.

EnsembleS was designed to incorporate the concepts of Multiparty session types with actor-based language Ensemble; this however had several implementation artifacts as mentioned in 2.1.4. Figure 4.3 exhibits the EnsembleS code for a DNS client [11]. We demonstrate the structural differences in the two programs and observe how the implementation challenges are prevented in Mini-EnsembleS.
Lines 1-7 in the EnsembleS program in figure 4.3 exhibit the unidirectional channels required to establish communication in the Client interface. A channel is used for each of the message types; 2 channels for sending strings to root and zone servers (lines 2,4) and similarly, 2 channels to receive strings (lines 3,5). Channels accepting a choice type from root and zone servers are defined at line 6 and line 7 respectively. Communication using an additional message type will require its own channel to be defined here in the client interface. Mini-EnsembleS eliminates this superfluous requirement in its straightforward syntax as shown in 4.4. Syntax for linking (line 26,37) and unlinking (33, 45, 49, 53) and the discovery process as well (line 24, 36) have been simplified.

In conclusion, deriving observations from the given examples and the produced results, our implementation can successfully type check the input code for Mini-EnsembleS programs, based on the core calculus of EnsembleS language.

Chapter 5

Conclusion and Future Work

There is a pertinent and demanding need for static technology to adapt to the evolving dynamic environment system, as computers and associated machinery evolve. Software adaptation allows devices to discover, replace and communicate with other software components at run-time, which were not part of the original system design. EnsembleS is a session-typed actor-based language that supports safe adaptable applications using Multiparty Session Types with explicit connection actions. This language builds directly on top of Ensemble, an imperative language designed for concurrent and distributed programming; this introduces various implementation artefacts, which makes the core features of the developed language abstruse. The objective of this project is to implement a typechecker for a smaller, clean language - Mini-EnsembleS, which expresses the core calculus and fundamental concepts in a clear and concise manner.

The implementation involves designing a concrete syntax to interpret the core calculus and a parser to translate the input code into an Abstract Syntax Tree. This tree is fed into the developed typechecker, which is the central component of our type system, and any potential type errors are detected and displayed. The rules of this type system are derived from the core calculus, which is based on the actor paradigm and MPST theory. Haskell is used for this implementation because of its support for algebraic data types, features like pattern matching, recursion and monads, and the nature of functional programming paradigm to express concepts in an algorithmic and pure manner. Evaluation of the typechecker is done using classical examples like Ping-Pong scheme and an Online-Store, and based on the capability of our type system to detect and prevent defined errors. We also present a comparison of input source codes between EnsembleS and Mini-EnsembleS to demonstrate the ease in implementation and eloquence in conveying the core concepts.

The type checker can provide a static type checking to guarantee a set of safety properties. A runtime system has to be developed to allow the programs to be executed. An exception handling mechanism must be integrated as well to correct the type errors. Mini-EnsembleS is specifically designed to covey the fundamental theory and calculus of EnsembleS. A more expressive language, which includes higher level of constructs, will be required for real-world deployment. Similarly, the processing and signalling of parsing errors and the detected type errors has to be more informative and ergonomic. Flexible exception handling or other non-traditional MPST theories can be developed, tested and experimented with using this base model, specifically designed for adaptation. The developed model aims to serve as a basis for future extension and experimentation.

Appendix A

Appendix

A.1 Concrete Syntax

In this section, we discuss the concrete syntax developed to implement the core calculus discussed in section § 2.2. Since the calculus is minimalistic, the syntax developed is very clean and concise and resembles the calculus to ensure readability and adaptability.

A Mini-EnsembleS program can be represented as a set of $(\overrightarrow{TA}, \overrightarrow{D}, \overrightarrow{P}, M)$, where $\overrightarrow{TA}$ is a collection of session type aliases. Type aliases provide alternative names for defining session types. Since interactions in multiparty session types can be quite complicated and go beyond a simple call/return expression, type aliases provide a convenient alternative to referring to the rather lengthy session type definitions. $\overrightarrow{D}$ is a list of all the Actor Definitions in the program, and $\overrightarrow{P}$ is the list of protocols specified, mapping participant roles to respective session types. M is the initial computation term in the 'boot' clause for initiating the actor communications.

Programs written in Mini-EnsembleS must follow a strict specification as to the sequence of declarations. Use of CamelCase is encouraged for the naming convention of all strings and variables in the syntax.

All type aliases must be defined together at the start of the program. A minimal syntax for the same is: `type <SessionTypeName> = <SessionType>`. Care must be taken that the identifier of the session type i.e., <SessionTypeName> must be capitalized.

This must be followed by all the actor definitions, each of the syntax: `actor <actorName> follows (<sessionType>) <computation>`. <actorName> can be any valid string beginning with a character, without any spaces or special symbols in the name.

All the protocol definitions should be defined next. The syntax for a protocol is `<role> : <sessionType>`. These can simply be defined together one after another.

Following all definitions, a Mini-EnsembleS must have a 'boot' clause for the initial computation term. The syntax for the same looks like this: `boot <computation>`

A.1.1 Concrete Syntax for Types and Terms

Values written in quotes will be treated as strings, for instance, "hello world" will be parsed as `EString "hello world"`. Integers such as 123 will be translated directly like `EInt 123`; case-insensitive input of boolean values such as true/True or false/False will be parsed as `EBool True` and `EBool False` respectively. Any other input for value will be treated as a variable name and interpreted as `EVar <varName>`. Actions defined in the calculus should have one of the following syntaxes:

1. `return <value>`
2. `continue <label>`
3. `raise`
4. `<value> == <value>`
5. `<value> /= <value>`
6. `new <actorName>`
7. `self`
8. `replace <value> with <behaviour>`
9. `discover <sessionType>`
10. `connect <label>(<value>) to <value> as <role>`
11. `accept from <role> case <label>(<value>) -> <computation> ..`
12. `accept <label><value> from <role>; <computation>`
13. `send <label>(<value>) to <role>`
14. `receive from <role> case <label>(<value>) -> <computation> ..`
15. `receive <label><value> from <role>; <computation>`
16. `wait <role>`
17. `disconnect from <role>`

Computations defined in the syntax should be one of the following syntaxes:

1. `let <binder> <= <compuation> in <compuation>`
2. `comp1; comp2`
3. `try <action> catch <compuation>`
4. `if <action> then <computation> else <computation>`
5. `<label> :: <computation>`
6. `<action>`

A.1.2 Concrete Syntax for Session action and Session types

Types: A type is generally passed as payload in the session type. This is used to validate that a particular communication action in a term has the matching value type in its session action. A type can be defined as one of the following: `Pid(<sessionType>)`, `unit`, `string`, `int`, or `bool`.

Session Actions: The generic form of communication actions is : `<role> <sessionAction> <label> <type>`. `<sessionAction>` could be either ! (send), !! (connect), ? (receive), ?? (accept). A session action could also be a wait action defined as `#<role>`.

Session Type: A session type can be a choice type of actions, $\Sigma_{i \in I}(\alpha_i \,.\, S_i)$ which can be written syntactically as:
$(\text{sessionAction}_1.\text{sessionType}_1) + .. + (\text{sessionAction}_n.\text{sessionType}_n)$, where each sessionAction_j is a valid session action. The '.' operator is used for sequencing session actions and session types. A recursive session type $\mu X.S$ can be written as `rec X . S`. A session type can also be a recursion variable `X` inside the continuation; care must be taken that recursion variables begin with a small-case character in the syntax. A session type can be a disconnection action in which case the syntax is `##<role>`. A finished session type is `end`. In addition to these, a session type can also be identified using it's type alias. A design decision dictates that this must be capitalized string name, represented as `<sessionTypeName>`.

This concludes the syntax description for our language. Further extensions or alterations can be easily made to this syntax; the objective during the design process was to provide a syntax to support the fundamental calculus. Hence, the implementation is flexible in order to adapt and accommodate any changes or extensions to the concrete syntax.

A.2 Abstract Syntax Tree

The abstract syntax tree implemented for the Mini-Ensemble language is displayed in A.2.

Following the calculus § 2.2, actors, roles, binder variables, and labels are processed as simple strings using the keyword 'type'. 'type' keyword in Haskell is used to create a synonym for an already existing type, in this case String. Using the data keyword, we define new data types with multiple value constructors, specifying the different values a specific data type can have. Line 8 defines a Behaviour which could be either a Computation or the stop term. Lines 10-14 allow for a Type to be one of EPid(SessionType), String, Int, Bool or Unit Types. Similarly, values, represented as EValue in the tree, can be a variable, integer, boolean, or the unit value, expressed as "()". Lines 24-38 define all the actions described in the calculus. Actions EAccept (line 32) and EReceive (line 34) make use of the type Choices defined as a list of tuples, each tuple describing a choice with a label, binder variable, and a continuation. Lines 37 and 38 are the comparison actions EEquality (V == W) and EInequality (V $\neq$ W) respectively.

Lines 40-45 each define a form of Computation; EAssign takes in a computation of the construct $\textbf{let}\, x \Leftarrow M\, \textbf{in}\, N$. Line 45 denotes the sequential operation i.e. M; N for computations. Lines 48-52 describe the different session actions - send, connect, receive, accept and wait. Line 56 defines Session types using the value constructor SAction and a list of different Session actions and their corresponding session types. Line 61 defines a value constructor STypeIdentifier which is used to refer to the type alias, if defined, for that session type. Together, lines 56-61 define a Session type. Line 66 is used to define an actor definition, taking in an actor's name, its session type, and the computation behavior it follows. A Protocol is defined as a mapping between a Role and its Session type, as seen in line 68. Lastly, a Program is defined as a 4-tuple entity, consisting of a list of type aliases, actor definitions, protocols, and an initial computation term. The formal calculus is broken down in this manner for our implementation.

```haskell
type Binder = String
-- Actor Name
type Actor = String
-- Roles
type Role = String

-- Behaviours
data Behaviour = EComp Computation |
    EStop
-- Types
data Type = EPid SessionType
  | TString
  | TInt
  | TBool
  | Unit
-- Values
data EValue = EVar String
  | EInt Int
  | EBool Bool
  | EUnit

type Label = String
type Choices = [(Label, EValue,
    Computation)]
-- Actions
data EAction = EReturn EValue
  | EContinue Label
  | ERaise Type
  | ENew Actor
  | ESelf
  | EReplace EValue Behaviour
  | EDiscover SessionType
  | EConnect Label EValue EValue Role
  | EAccept Role Choices
  | ESend Label EValue Role
  | EReceive Role Choices
  | EWait Role
  | EDisconnect Role
  | EEquality EValue EValue
  | EInequality EValue EValue
-- Computations
data Computation = EAssign Binder
    Computation Computation
  | ETry EAction Computation
  | ERecursion Label Computation
  | EAct EAction
  | ECondition EValue Computation
    Computation
  | ESequence Computation Computation

-- Session Actions
data SessionAction = SSend Role Label
    Type
  | SConnect Role Label Type
  | SReceive Role Label Type
  | SAccept Role Label Type
  | SWait Role
type RecursionVar = String
type SessionTypeName = String
-- SessionTypes
data SessionType = SAction [(
    SessionAction, SessionType)]
  | SRecursion RecursionVar SessionType
  | SRecursionVar RecursionVar
  | SDisconnect Role
  | SEnd
  | STypeIdentifier SessionTypeName

-- TypeAlias
newtype TypeAlias = SessionTypeAlias
    SessionType SessionType
-- Actor Definition
newtype ActorDef = EActorDef Actor
    SessionType Computation
-- Protocol
newtype Protocol = Protocol Role
    SessionType
-- Program
type Program = ([TypeAlias], [ActorDef
    ], [Protocol], Computation)
```

Figure A.1: Abstract Syntax Tree for Mini-EnsembleS

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0262010925.

[2] Joe Armstrong. "Erlang". In: *Commun. ACM* 53.9 (Sept. 2010), pp. 68–75. ISSN: 0001-0782. DOI: 10.1145/1810891.1810910. URL: https://doi.org/10.1145/1810891.1810910.

[3] Luca Cardelli. "Type Systems". In: *ACM Comput. Surv.* 28.1 (Mar. 1996), pp. 263–264. ISSN: 0360-0300. DOI: 10.1145/234313.234418. URL: https://doi.org/10.1145/234313.234418.

[4] Massimo Cossentino et al. "A Comparison of the Basic Principles and Behavioural Aspects of Akka, JaCaMo and Jade Development Frameworks". In: *WOA*. 2018.

[5] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. "43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties". In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 31–40. ISBN: 9781450346399. DOI: 10.1145/3001886.3001890. URL: https://doi.org/10.1145/3001886.3001890.

[6] Ugo de'Liguoro and Luca Padovani. "Mailbox Types for Unordered Interactions". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Ed. by Todd Millstein. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 15:1–15:28. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.15. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9220.

[7] Ducklang. *Designing a Programming Language: I*. URL: http://ducklang.org/designing-a-programming-language-i#designing-a-programming-language-i. (accessed: 010.12.2021).

[8] Simon Fowler, Sam Lindley, and Philip Wadler. "Mixing Metaphors: Actors as Channels and Channels as Actors". English. In: *The 31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Leibniz International Proceedings in Informatics (LIPIcs). 31st European Conference on Object-Oriented Programming, ECOOP 2017 ; Conference date: 18-06-2017 Through 23-06-2017. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, June 2017, pp. 1–28. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.11. URL: https://2017.ecoop.org/track/ecoop-2017-papers.

[9] The University of Glasgow 2001. *Data.Either*. URL: https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Either.html. (accessed: 010.12.2021).

[10] Paul Harvey and Joseph Sventek. “Adaptable Actors: Just What The World Needs”. In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 22–28. ISBN: 9781450351539. DOI: 10.1145/3144555.3144559. URL: https://doi.org/10.1145/3144555.3144559.

[11] Paul Harvey et al. “Multiparty Session Types for Safe Runtime Adaptation in an Actor Language (Extended version)”. In: *ArXiv* abs/2105.06973 (2021).

[12] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *J. ACM* 63.1 (Mar. 2016). ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://doi.org/10.1145/2827695.

[13] Raymond Hu and Nobuko Yoshida. “Explicit Connection Actions in Multiparty Session Types”. In: *FASE*. 2017.

[14] Mark Karpov. *Megaparsec tutorial*. URL: https://markkarpov.com/tutorial/megaparsec.html. (accessed: 08.12.2021).

[15] Paul Blain Levy, John Power, and Hayo Thielecke. “Modelling environments in call-by-value programming languages”. English. In: *Information and Computation* 185.2 (Sept. 2003), pp. 182–210. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00088-9.

[16] Paolo Martini and Daan Leijen. *megaparsec: Monadic parser combinators*. URL: https://hackage.haskell.org/package/megaparsec. (accessed: 08.12.2021).

[17] Andriy Palamarchuk Michael Weber Jeff Newbern. *Control.Monad.Except*. URL: https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html. (accessed: 010.12.2021).

[18] Barry Porter et al. “REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 333–348. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter.

[19] Alceste Scalas and Nobuko Yoshida. “Less is More: Multiparty Session Types Revisited”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290343. URL: https://doi.org/10.1145/3290343.

[20] Nobuko Yoshida and Lorenzo Gheri. “A Very Gentle Introduction to Multiparty Session Types”. In: *16th International Conference on Distributed Computing and Internet Technology*. Vol. 11969. LNCS. Springer, 2020, pp. 73–93. DOI: 10.1007/978-3-030-36987-3_5.

[21] Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. “Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types”. In: Sept. 2021, pp. 18–35. ISBN: 978-3-030-86592-4. DOI: 10.1007/978-3-030-86593-1_2.

[22] Nobuko Yoshida et al. “The Scribble Protocol Language”. In: *TGC*. 2013.