# Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

Théa Johnson

April 20, 2016

# Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Théa Johnson

April 20, 2016

# Formal Verification of Session Types

Théa Johnson, B.A.
University of Dublin, Trinity College, 2016

Supervisor: Dr. Vasileios Koutavas

This report details the development of an understanding of session types starting from a very basic level as well as the implementation of a design for a behaviour checker. This checker can takes the behaviours of the communication of the input code and can confirm if the input program can communicate correctly or not.

# Contents

# Chapter 1

# Introduction

## 1.1 Project Objectives

The aims of this project are to investigate the formal verification of session types, specifically in relation to the paper Type-Based Analysis for Session Inference [1]. Also to implement a behaviour checker based on the designs described in this paper using OCaml, Menhir and OCamllex.

## 1.2 Summary of Report

In this report first, in this chapter, a brief background to the area is given. The following chapter then gives an overview and explanation of the system proposed in the paper Type-Based Analysis for Session Inference [1]. The third chapter details the implementation completed over the course of this project. The final chapters cover some detailed examples of how programs are dealt with in this system and an evaluation of the system. An outline of the system can be seen in (fig. 1.1).

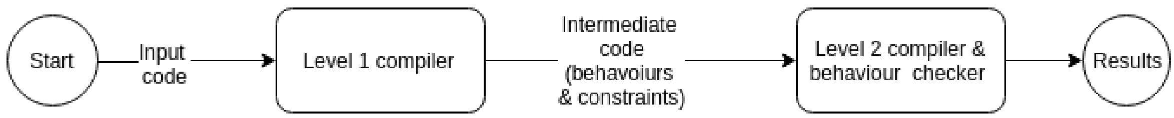


Figure 1.1: System Overview

## 1.3 Motivation

The modern world is growing increasingly dependent on distributed systems, changing the historical approach to computing dramatically. In order for modern society to function it is important that these systems communicate correctly and that when proposing or introducing new systems it can be shown that they will communicate correctly under all circumstances.

Modern programming languages support data types. These allow us to use verification techniques to show that the program will run as expected on all forms of input. A similar system of types could be used for communication over distributed systems. Ideally a type system for communication would be embedded into languages in a similar fashion to the data type systems of modern languages.

This is the system that is proposed with session types. These types can specify the style of communication expected (in general terms send or receive) as well as the type of data that is expected. A detailed article on the importance of implementation of this style of system can be found in the BETTY-EU-Research magazine article [2].

### 1.3.1 Motivating example

A simple example of a swap system is given here and a short discussion of how it would interact with this system. A more in depth discussion of the use of this system to confirm that this program works as expected can be found in chapter 4.

```
let fun coord(_) =
  let val p1 = acc swp ()
      val x1 = recv p1
      val p2 = acc swp ()
      val x2 = recv p2
  in send p2 x1; send p1 x2;
     coord ()
in spawn coord;
```

```
let fun swap(x) =
  let val p = req swp ()
  in send p x; recv p
in spawn (fn _ => swap 1);
   spawn (fn _ => swap 2);
```

Here there are two swap processes and one coordinator process. The coordinator accepts two connections on the swp channel. After accepting each connection it receives a value over it. Finally these values are swapped and sent over the connections. In the swap function a connection is requested on the swp channel and a value is sent over it.

A value is then received on this connection.

In its current state this program is typable and can communicate since the swap processes will both send integers that will then be sent to the alternate process. In fact the system proposed in (Spacasassi & Koutavas) [1] can infer the session types of the connections established in this example. Both p1 and p2 would have the type $?Int.!Int.end$ and p would be of the form $!Int.?Int.end$. In this syntax the !'s represent values being sent and the ?'s values received.

We can see however that the swap processes will not be able to deduce which connection they will get when they request a connection. This means that the use of p1 and p2 must be the same. If this were not the case and say a value was sent on p2 first then this program could not work. If this were the case the system discussed in this report would spot that this was a problem when the behaviour checker was run since the connection protocols would not match with the behaviours of the code.

An example of a program that would fail would be if we were to replace the above swap function with this one:

```
let fun swap(x) =
  let val p = req swp ()
  in recv p; send p x
in spawn (fn _ => swap 1);
   spawn (fn _ => swap 2);
```

In this case the protocol of p would be $?int.!int.end$ which is the same as that of p1 and p2. It is clear that these protocols are not compatible. The behaviour checker described in this report will reject this version of the program as been incorrect due to this.

## 1.4 Current Work

This area is currently been researched by multiple groups. However it is currently not used in real world systems to any great extent. To date systems have been developed for applying session type disciplines to functional languages, object oriented languages and operating systems.

## 1.5 Background

Traditional type systems embedded into programming languages focus on the computations and what they should produce. These systems will quickly inform a programmer if an attempt is made to apply some operation to a nonsensical data type. This level of guidance does not currently exist for communication protocols. While languages may help with the format of messages ensuring that the ordering is correct is left to rely on manual testing. Session type disciplines aim for embedded session types that can describe the sequence of messages as well as the type of the messages transmitted on communication channels. Then, since session types will describe the protocols of channels, verification techniques can be used to ensure that processes will abide by these protocols and more guidance can be given to programmers writing programs for distributed systems.

### 1.5.1 Main session type approaches

The main approaches to session types, according to (Hüttle et al.) [3] are detailed in the following section.

*Session types* are usually associated with binary communication channels where the two ends using the channel view the endpoints as complementary types. Static type checking can then be used to ensure that the communications on the channel abide by the protocol specified.

*Multiparty session types* extend binary session types to allow for more than two processes to communicate.

*Conversation session types* unify local and global multiparty types and allow for an unspecified number of processes to communicate over the channel.

*Contracts* focus on general theory to confirm that communications follow the specified abstract description of input/output actions.

# Chapter 2

# Summary of the Paper "Type Based Analysis for Session Inference"

## 2.1 Overview

This paper[1] proposes a system for a design approach to Binary Session Types which uses effects. For this a high level language is developed where communication protocols can be programmed and statically checked. In the paper the approach is applied to $ML_s$, a core of the language ML with session communication.

The approach suggested separates traditional typing and session typing using a two level system. The first level uses a type and effect system, which is an adaptation of the one developed by (Amtoft, Neilsen and Neilsen, 1999 )[4]. The types allow for a clear representation of program properties and enable modular reasoning. The effects are essentially behaviour types that record the actions of interest, relevant to the communication, that will occur at run time. From this system a complete behaviour inference algorithm is obtained. This derives a behaviour for a program providing it respects ML types. At this level the programs are typed against both an ML type and a behaviour. Session protocols are not considered here and so endpoints (see 2.3) are given the type $ses^{\rho}$ instead.

The second level checks the behaviour to see that it complies with the session types of both channels and endpoints. In performing this check the operational semantics are used (see fig. 2.4).

This level is inspired by the work done by Castagna et al. [5]. In their system session based interaction is established on a public channel, once established the parties continue to communicate on a private channel. Messages are exchanged on the private channel according to a given protocol. Internal and external choices are also required to implement control. Internal choices are when the decision is made autonomously by a process and external choices occur when a decision is based entirely on messages received.

This level ensures that sessions respect the order of communication and message types described by the session type of the channel. It also ensures partial lock freedom due to the stacked interleaving of sessions.

One of the most appealing aspects of the session type discipline proposed here is that it allows for complete inference of the session type from the behaviours. When this is combined with behaviour inference from level 1 a method for complete session type inference without programmer annotations is achieved.

The two levels of the system only interact through behaviours. This allows for the development of front ends for different languages and back ends for different session disciplines and to combine the two to cover an extensive selection of requirements. The behaviour checker implementation detailed in this report (see 3.4) can be used with any implementation of the first level provided that its output is of the correct format.

This system, since it is run at compile time, is forced to over-approximate the runtime behaviour of the program. This, combined with the fact that it will not return false positives implies that it is possible for the system to reject a program that would in fact communicate correctly.

The behaviour checker implemented as part of this project is an implementation of the second level of this system (without inference). The first level of this system was implemented by the authors of the paper.

## 2.2 The First Level

At this level the type and effect system of (Amtoft, Neilsen and Neilsen, 1999) [4] is extended to session communications in $ML_s$. The type and effect system consists of

$$
\begin{array}{rl}
\textbf{Variables:} & \alpha(\textbf{Type}) \quad \beta(\textbf{Behaviour}) \quad \psi(\textbf{Session}) \quad \rho(\textbf{Region}) \\
\textbf{T. Schemas:} & TS ::= \forall(\vec{\alpha}\vec{\beta}\vec{\rho}\vec{\psi} : C).\, T \qquad\qquad \textbf{Regions:}\ r\ ::=\ l \mid \rho \\
\textbf{Types:} & T ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid T \times T \mid T \xrightarrow{\beta} T \mid \mathsf{Ses}^{\rho} \mid \alpha \\
\textbf{Constraints:} & C ::= T \subseteq T \mid cfd(T) \mid b \subseteq \beta \mid \rho \sim r \mid c \sim \eta \mid \overline{c} \sim \eta \mid \eta \bowtie \eta \mid C, C \mid \epsilon \\
\textbf{Behaviours:} & b ::= \beta \mid \tau \mid b\,;b \mid b \oplus b \mid \mathsf{rec}_{\beta}\, b \mid \mathsf{spawn}\, b \mid \mathsf{push}(l : \eta) \\
& \quad \mid\ \rho!T \mid \rho?T \mid \rho!\rho \mid \rho?l \mid \rho!L_i \mid \underset{i \in I}{\&}\{\rho?L_i\,; b_i\} \\
\textbf{Type Envs:} & \Gamma ::= x : TS \mid \Gamma, \Gamma \mid \epsilon
\end{array}
$$

Figure 2.1: Syntax of types, behaviours and constraints

constructs of judgements. These judgements are of the form $C; \Gamma \vdash e : T \triangleright b$. Where $C$ represents the constraint environment which is used to relate type level variables to terms and enable session inference and $\Gamma$ represents the type environment which is used to bind program variables to type schemas. To read this judgement we would say that expression $e$ has type $T$ and behaviour $b$ under type environment $\Gamma$ and constraint environment $C$.

In the system designed in this paper [1] an $ML_s$ expression can have either a standard ML type or a session type. Session types are of the form $ses^{\rho}$ where $\rho$ is a static approximation of the location of the endpoint. Functional types have an associated behaviour $\beta$ and type variables, $\alpha$, are used for ML polymorphism.

Polymorphism is extended with type schemas. These are of the form $\forall(\overrightarrow{\gamma} : C_0).T$ where $\gamma$ is a list made up of some combination of type ($\alpha$), behaviour ($\beta$), region ($\rho$) and session ($\psi$) variables and $C_0$ represents the constraint environment that imposes constraints on the quantified variables.

### 2.2.1 Syntax of types, behaviours and constraints

Of the types detailed in 2.1 behaviours and constraints are the most relevant to the implementation discussed in this report ( see chapter 3). Behaviours can take several straightforward forms; a behaviour variable ($\beta$), the behaviour with no effect($\tau$), a sequence of behaviours ($b; b$) or a choice in behaviours ($b \oplus b$). They can also take more complicated forms including $rec_{\beta}b$ for recursive behaviour. In this case, and in the case of $spawn \quad b$, the body of the recursion must be confined (it must not affect endpoints that are already open and it must consume all endpoints it opens). Behaviour can also input or output types ($\rho!T, \rho?T$), delegate and resume endpoints ($\rho!\rho, \rho!l$), select from an internal choice $\rho!L_i$ and offer external choice $\underset{i \in I}{\&}\{\rho?L_i; b_i\}$.

Constraints can specify that types are subtypes of another type ($T \subseteq T$), or that they are confined ($cfd(T)$) which means that values of this type have no communication effect. They also specify which behaviours behaviour variables can act as ($b \subseteq \beta$). Region constraints ($\rho \sim r$) link region variables to other region variables or to locations. Channel constraints ($c \sim \eta, \bar{c} \sim \eta$) specify the link between channels and their endpoints. The duality constraint ($\eta \bowtie \eta$) states that two endpoints are complementary. Sets of constraints can also be specified ($C, C$).

Labels ($l$) are static compile time approximations of the runtime channel endpoint locations. Since the system described here is run at compile time it has to over-approximate the runtime behaviours.

### Type schemas, locations and region variables

Through the constraint environment ($C$) region constraints specify links between region variables ($\rho$) and other region variables or labels. These region constraints are produced during pre-processing and they identify the textual source of endpoints. The labels $l$ specify the locations in the input code at which endpoints are generated.

For example if a connection on the private channel c is requested ($req - c^l$) then l is the location in the code it is called from. This means that if this were to be called in, for example, a for loop we would have multiple instances of endpoints related to l. This is a limitation of this system.

If we have an expression with type $Ses^\rho$ which evaluates to $p^l$ then a region constraint must exist to link $l$ to $\rho$. This would take the form $C \vdash \rho \sim l$ which is to say that $\rho$ and $l$ are linked under constraint environment $C$. This tells us that $p$ was generated from the location in the code referenced by $l$.

If we were to look up this location we would find one of $req{-}c^l$, $acc{-}c^l$ or $resume{-}c^l$ where $c$ references the private channel on which the communication will take place. These primitive functions are typed by the rule TConst given in (fig. 2.3).

The type schemas of these primitives are given in (fig. 2.2). In $req{-}c^l$, for example, a new session is started on the static endpoint $l$. In order for it to be type-able $C$ must

$$\begin{aligned}
req - c^l :&\forall(\beta\rho\psi : push(l:\psi) \subseteq \beta, \rho \sim l, c \sim \psi).Unit \xrightarrow{\beta} Ses^\rho \\
acc - c^l :&\forall(\beta\rho\psi : push(l:\psi) \subseteq \beta, \rho \sim l, \bar{c} \sim \psi).Unit \xrightarrow{\beta} Ses^\rho \\
ressume - c^l :&\forall(\beta\rho\rho' : \rho?\rho' \subseteq \beta, \rho' \sim l).Ses^\rho \xrightarrow{\beta} Ses^{\rho'} \\
recv :&\forall(\alpha\beta\rho : \rho?\alpha \subseteq \beta cfd(\alpha)).Ses^\rho \xrightarrow{\beta} \alpha \\
send :&\forall(\alpha\beta\rho : \rho!\alpha \subseteq \beta cfd(\alpha)).Ses^\rho \times \alpha \xrightarrow{\beta} Unit \\
deleg :&\forall(\alpha\rho\rho' : \rho!\rho' \subseteq \beta).Ses^\rho \times Ses^{\rho'} \xrightarrow{\beta} Unit \\
sel - L :&\forall(\alpha\rho : \rho?L \subseteq \beta).Ses^\rho \xrightarrow{\beta} Unit
\end{aligned}$$

Figure 2.2: Type Schemas

contain its effect which is $push(l : \psi) \subseteq \beta$. In the stack frame $\psi$ is the session variable that represents the session type of endpoint $l$. This representation is used since session types are only considered in the second level. $C$ must also record that the region variable $\rho$ is linked to $l$ and that the 'request' endpoint of $c$ has session type $\psi$.

The remaining type schemas are read in a similar way.

### 2.2.2 Type and effect system

The rules for the type and effect system proposed are given in (fig. 2.3). These consist of the judgements described above and the requirements for the rule. These rules say that if we have a judgements of the form given above the line and if the requirements beside the rule (if they exist) are met then the judgement below the line will be valid.

For example the rule TSub states that if we have constraint environment $C$, type environment $\Gamma$, expression $e$ of type $T$ with associated behaviour $b$ and we also know that constraint environment $C$ contains constraints telling us that $T$ is a functional subtype of $T'$ and the $b$ is a sub-behaviour of $\beta$ then we can say that under the same type and constraint environments the expression $e$ can be said to have type $T'$ and behaviour $\beta$.

Of the other rules TLeft, TVar, TIf, TConst, TApp, TFun, TSpawn and TPair are used to perform standard type checking of sequential and non-deterministic behaviour.

TIns and TGen are used to extend the instantiation and generalisation rules of ML.

TRec ensures that in the case of recursion the communication effect of the body of the recursion is confined. Meaning that it will not effect any endpoints already open when called and will consume all end points opened during its execution.

TEndP ensures that if we have an expression with the type of a session endpoint with associated region and this expression evaluates to a value associated with a location then there exists a link between the region and the location in the constraint environment.

TConst types primitive functions such as $req - c^l$ (see sec. 2.2.1).

$$\textsc{TPair}\ \frac{C;\ \Gamma \vdash e_1 : T_1 \triangleright b_1 \quad C;\ \Gamma \vdash e_2 : T_2 \triangleright b_2}{C;\ \Gamma \vdash (e_1, e_2) : T_1 \times T_2 \triangleright b_1 \,;b_2}$$

$$\textsc{TVar}\ \frac{}{C;\ \Gamma \vdash x : \Gamma(x) \triangleright \tau}$$

$$\textsc{TIf}\ \frac{C;\ \Gamma \vdash e_1 : \mathsf{Bool} \triangleright b_1 \quad C;\ \Gamma \vdash e_i : T \triangleright b_i\ {}_{(i\in\{1,2\})}}{C;\ \Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : T \triangleright b_1 \,; (b_2 \oplus b_3)}$$

$$\textsc{TConst}\ \frac{}{C;\ \Gamma \vdash k : \mathit{typeof}(k) \triangleright \tau}$$

$$\textsc{TApp}\ \frac{C;\ \Gamma \vdash e_1 : T' \xrightarrow{\beta} T \triangleright b_1 \quad C;\ \Gamma \vdash e_2 : T' \triangleright b_2}{C;\ \Gamma \vdash e_1\, e_2 : T \triangleright b_1 \,; b_2 \,; \beta}$$

$$\textsc{TFun}\ \frac{C;\ \Gamma, x : T \vdash e : T' \triangleright \beta}{C;\ \Gamma \vdash \mathsf{fn}\, x \Rightarrow e : T \xrightarrow{\beta} T' \triangleright \tau}$$

$$\textsc{TMatch}\ \frac{C;\ \Gamma \vdash e : \mathsf{Ses}^\rho \triangleright b \quad C;\ \Gamma \vdash e_i : T \triangleright b_i\ {}_{(i\in I)}}{C;\ \Gamma \vdash \mathsf{case}\, e\, \{L_i : e_i\}_{i\in I} : T \triangleright b \,; \underset{i\in I}{\&}\{\rho?L_i \,; b_i\}}$$

$$\textsc{TEndp}\ \frac{}{C;\ \Gamma \vdash p^l : \mathsf{Ses}^\rho \triangleright \tau}\ \boxed{C \vdash \rho \sim l}$$

$$\textsc{TLet}\ \frac{C;\ \Gamma \vdash e_1 : \mathit{TS} \triangleright b_1 \quad C;\ \Gamma, x : \mathit{TS} \vdash e_2 : T \triangleright b_2}{C;\ \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : T \triangleright b_1 \,; b_2}$$

$$\textsc{TSub}\ \frac{C;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : T' \triangleright \beta}\ \boxed{\begin{array}{l} C \vdash T <: T' \\ C \vdash b \subseteq \beta \end{array}}$$

$$\textsc{TSpawn}\ \frac{C;\ \mathit{confd}_C(\Gamma) \vdash e : \mathsf{Unit} \xrightarrow{\beta} \mathsf{Unit} \triangleright b}{C;\ \Gamma \vdash \mathsf{spawn}\, e : \mathsf{Unit} \triangleright b \,; \mathsf{spawn}\, \beta}$$

$$\textsc{TRec}\ \frac{C;\ \mathit{confd}_C(\Gamma), f : T \xrightarrow{\beta} T', x : T \vdash e : T' \triangleright b}{C;\ \Gamma \vdash \mathsf{fun}\, f(x) = e : T \xrightarrow{\beta} T' \triangleright \tau}\ \boxed{\begin{array}{l} C \vdash \mathit{confd}(T, T') \\ C \vdash \mathsf{rec}_\beta\, b \subseteq \beta \end{array}}$$

$$\textsc{TIns}\ \frac{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b}{C;\ \Gamma \vdash e : T\sigma \triangleright b}\ \boxed{\begin{array}{l} \mathit{dom}(\sigma) \subseteq \{\vec{\gamma}\} \\ \forall(\vec{\gamma} : C_0).\, T \text{ is solvable from } C \text{ by } \sigma \end{array}}$$

$$\textsc{TGen}\ \frac{C \cup C_0;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b}\ \boxed{\begin{array}{l} \{\vec{\gamma}\} \cap \mathrm{fv}(\Gamma, C, b) = \emptyset \\ \forall(\vec{\gamma} : C_0).T \text{ is WF, solvable from } C \end{array}}$$

Figure 2.3: Type and Effect system for $ML_s$ Expressions

## 2.3 The Second Level

At this level session types are considered. Theses take the form:

$$\eta ::= end \mid !T.\eta \mid ?T.\eta \mid !\eta.\eta \mid ?\eta.\eta \mid \bigoplus_{i \in I} \{L_i : \eta_i\} \mid \underset{i \in (I_1, I_2)}{\&} \{L_i : \eta_i\} \mid \psi$$

Either communication on the endpoint is finished ($end$) or more communications are going to take place. These include sending or receiving a confined type ($!T.\eta, ?T.\eta$), delegating by sending one endpoint over another ($!\eta.\eta$) or resuming an endpoint ($?\eta.\eta$). Non deterministic selection involves selecting a label $L_i$, selected by the selection behaviour from a list $I$, and then following the selected session type $\eta_i$. External choice ($\underset{i \in (I_1, I_2)}{\&} \{L_i : \eta_i\}$) is also supported.

### 2.3.1 Abstract interpretation semantics

In these semantics (fig. 2.4) $\Delta$ represents the stack which consists of stack frames made up of a label and a session type, $c$ represents the channel, $C$ is the set of constraints. A behaviour and the current stack considered to see if they match one of the rules. These rules then describe what actions are to be taken.

The stack, mentioned above, is used to avoid deadlock between the processes when this would be due to cyclic dependencies as is the case in the erroneous example discussed in the introduction (see 1.3.1).

For example the REC Rule states that if the current behaviour is recursion then we must check that the behaviour associated with that recursion will follow these rules, starting with an empty stack, to end in a state with the empty stack and the behaviour $\tau$. Another constraint is that any behaviour constraints in the constraint environment of the form $rec_\beta b \subseteq \beta$ must be replaced with $\tau \subseteq \beta$ when checking the body of the recursion. If these constraints are met then the next behaviour to be checked is $\tau$ with the stack remaining unchanged. This ensures that recursive behaviour is confined, which is necessary for this system.

As for the other rules END removes a finished stack frame. BETA looks up behaviour variables and subs in the behaviours associated with them in the constraint environment. PLUS chooses a branch. PUSH adds a new frame to the stack given that the label has not previously been pushed to the stack. OUT and IN reduce the frame

| | | |
|---|---|---|
| END : | $(l : \mathsf{end}) \cdot \Delta \vDash b \rightarrow_C \Delta \vDash b$ | |
| BETA : | $\Delta \vDash \beta \rightarrow_C \Delta \vDash b$ | if $C \vdash b \subseteq \beta$ |
| PLUS : | $\Delta \vDash b_1 \oplus b_2 \rightarrow_C \Delta \vDash b_i$ | if $i \in \{1, 2\}$ |
| PUSH : | $\Delta \vDash \mathsf{push}(l : \eta) \rightarrow_C (l : \eta) \cdot \Delta \vDash \tau$ | if $l \notin \Delta.\mathsf{labels}$ |
| OUT : | $(l : !T.\eta) \cdot \Delta \vDash \rho!T' \rightarrow_C (l : \eta) \cdot \Delta \vDash \tau$ | if $C \vdash \rho \sim l,\ T' <: T$ |
| IN : | $(l : ?T.\eta) \cdot \Delta \vDash \rho?T' \rightarrow_C (l : \eta) \cdot \Delta \vDash \tau$ | if $C \vdash \rho \sim l,\ T <: T'$ |
| DEL : | $(l : !\eta_d.\eta) \cdot (l_d : \eta'_d) \cdot \Delta \vDash \rho!\rho_d \rightarrow_C (l : \eta) \cdot \Delta \vDash \tau$ | if $C \vdash \rho \sim l,\ \rho_d \sim l_d,\ \eta'_d <: \eta_d$ |
| RES : | $(l : ?\eta_r.\eta) \vDash \rho?l_r \rightarrow_C (l : \eta) \cdot (l_r : \eta_r) \vDash \tau$ | if $(l \neq l_r),\ C \vdash \rho \sim l$ |
| ICH : | $(l : \bigoplus_{i \in I}\{L_i : \eta_i\}) \cdot \Delta \vDash \rho!L_j \rightarrow_C (l : \eta_j) \cdot \Delta \vDash \tau$ | if $(j \in I),\ C \vdash \rho \sim l$ |
| ECH : | $(l : \&_{i \in (I_1, I_2)}\{L_i : \eta_i\}) \cdot \Delta \vDash \&_{j \in J}\{\rho?L_j ; b_j\} \rightarrow_C (l : \eta_k) \cdot \Delta \vDash b_k$ | if $k \in J,\ C \vdash \rho \sim l,$ $I_1 \subseteq J \subseteq I_1 \cup I_2$ |
| REC : | $\Delta \vDash \mathsf{rec}_\beta\, b \rightarrow_C \Delta \vDash \tau$ | if $\epsilon \vDash b \Downarrow_{C'}$, $C' = (C \backslash (\mathsf{rec}_\beta\, b \subseteq \beta)) \cup (\tau \subseteq \beta)$ |
| SPN : | $\Delta \vDash \mathsf{spawn}\, b \rightarrow_C \Delta \vDash \tau$ | if $\epsilon \vDash b \Downarrow_C$ |
| SEQ : | $\Delta \vDash b_1; b_2 \rightarrow_C \Delta' \vDash b'_1; b_2$ | if $\Delta \vDash b_1 \rightarrow_C \Delta' \vDash b'_1$ |
| TAU : | $\Delta \vDash \tau ; b \rightarrow_C \Delta \vDash b$ | |

Figure 2.4: Abstract Interpretation Semantics

at the top of the stack. DEL and RES are used to transfer endpoints. RES must be applied to a one frame stack to ensure that there are no two endpoints of the same session pushed to the same stack (this is to avoid deadlock). ICH offers internal choice of session types. ECH offers external choice. SPN ensures that spawned processes are confined.

## 2.4 Inference Algorithm

There are three inference algorithms used in the system proposed in this paper [1]. The first of these is used with the first level to infer functional types and communication effects. The remaining two are used with the second level to infer session types from the abstract interpretation rules (fig. 2.4) and the duality requirement.

The Duality Requirement states that a constraint environment $C$ is valid if there exists a substitution $\sigma$ of variables $\psi$ with closed session types, such that $C_\sigma$ is well formed and for all $(c \sim \eta), (\bar{c} \sim \eta') \in C_\sigma$ we have $C \vdash \eta \bowtie \eta'$

The first algorithm is an adaptation of the homonymous algorithm from Amtoft, Neilsen & Neilsen [4]. From expression $e$ it calculates its type, behaviour and constraint set. No session information is calculated.

The second algorithm infers a substitution and a refined set of constraints in such a way that the empty stack and behaviour $\tau$ will be reached when the rules from (fig. 2.4) are applied to a behaviour on which the substitution has been applied. Assuming that the communication is correct.

The third algorithm deals with the constraints relating to channels and generates duality constraints. As well as this it also checks these constraints.

# Chapter 3

# My Contribution

My work for this project is based on the second level. this includes the development of a text based syntax for the intermediate code (the code produced from the first level), a lexer and parser for this syntax and a program to verify the behaviours produced.

The combination of lexer and parser converts the input code, which consists of one behaviour and one constraint, into the types defined in OCaml to represent the behaviours and constraints. These are then returned and passed into the behaviour checker where the rules from (fig. 2.4) are be applied to verify that the program communicates correctly.

One limitation of the behaviour checker implementation that is worth noting is that it will not infer types and so these must be manually added to in the input code.

## 3.1 Text Based Syntax

### 3.1.1 Design

The syntax given in the paper can be seen in (fig. 2.1). In order to develop an easy to parse version of this syntax Greek letters have been removed, keywords or individual letters have been chosen to replace them. The final syntax is very similar to that given in (fig. 2.1). Substitutions for variables and labels are detailed in table 3.1. Keywords and the structure of the input can be found in the grammar detailed in 3.2.4.

| | | |
|---|---|---|
| Type Variables | $\alpha$ | [T][A-Z a-z 1-9]+ |
| Behaviour Variables | $\beta$ | [B][A-Z a-z 1-9]+ |
| Session Variables | $\psi$ | [S][A-Z a-z 1-9]+ |
| Region Variables | $\rho$ | [R][A-Z a-z 1-9]+ |
| Labels | l | $[A-Z a-z 1-9]+$ |
| channel | c | [C][A-Z a-z 1-9]+ |
| channelEnd | $\bar{c}$ | [T][A-Z a-z 1-9]+ ['] |

Table 3.1: Substitutions to the given syntax

## 3.2 Lexer and Parser

Initially Camlp4 [6] was considered for this implementation. This is a Pre-Processor-Pretty-Printer for OCaml. However it soon became clear that this was not the appropriate tool since this is designed to extend the existing syntax of OCaml and not for developing a new syntax structure.

### 3.2.1 Lexer

The lexer was designed using OCamlLex which is based on Lex, a lexical analyser generator. This takes a set of regular expressions and corresponding semantic actions and generates a lexical analyser from them. In this case the regular expressions are the keywords, labels, variables, etc. and the associated actions are either tokens that link to the parser or, in the case of the opening $, calls a second lexing function to deal with labels.

### 3.2.2 Parser

The parser implementation was done using Menhir [7]. This is a parser generator that is closely related to OCamlyacc which is, in turn, based on yacc. These generate LR(1) parsers which means that the input is parsed from the bottom up and that there is one character look ahead.

To generate the parser the tokens are declared first. If they consist of a default type this is specified; for example the token $\%token < string > LABEL$ states that LABEL consists of a string. The structure of the accepted input is then specified. With this the code that is to be invoked when this input is encountered is also stated.

The structure of the accepted input can be found in section 3.2.4.

An example of the sub grammar for regions from the parser can be seen here:

```
regionVar:
  | l=LABLE
    {Label l}
  | r= REG
    {RVar r}
```

### 3.2.3 Challenges

The main challenge encountered when designing the lexer and the parser was learning the correct usage of OCamlLex and Menhir. Having never used anything like this before when starting from scratch these took some time to get used to. The tutorials [8][9][10, Chapter 16] were helpful in gaining an understanding of the syntax and structure.

Another challenge encountered with the lexer was parsing the end of file. An unknown issue causes the lexer to state that it finds a syntax error at the final character of the input file. While this is misleading to the user and not entirely correct it does not effect the behaviour of the behaviour checker and so has been ignored.

Initially there were some errors with shift/reduce conflicts in the parser. This means that Menhir was unable to determine which of two or more rules should be applied in a situation and so was choosing to use one of them. These were caused by not initially including operator precedence for ';' and ',' meaning that the execution order of sequencing of behaviours and constraints was unclear. As well as this there were not initially brackets in the

$$rec < behaviourVariable > (< behaviour >)$$

rule which also caused an issue. It was unclear in the case

$$rec < behaviourVariable >< behaviour >; < behaviour >$$

if the recursive behaviour was in the sequence or if the sequence was the body of the recursive behaviour.

### 3.2.4 Grammar

The grammar was designed with usability in mind. The main entry point for the grammar is shown first. This states that the parser will read a behaviour followed by a constraint followed by the end of file. In the description of the grammar all keywords, brackets etc. are input as they are given in this section. Anything inside <> is either another type from the grammar or a variable the syntax of which is given in table 3.1.

$\langle parse\_behaviour \rangle$ ::= $\langle behaviour \rangle$ $\langle constraint \rangle$ EOF

#### Behaviour

The behaviour type in the grammar consists of multiple options. The final option, for example, represents the offer of external choice and we can see that it contains a list. This will call to a sub grammar that states that this list consists of comma separated values of another subtype (opt_feild).

$\langle behaviour \rangle$ ::= $\langle behaviourVariable \rangle$
| tau
| $\langle behaviour \rangle$ ; $\langle behaviour \rangle$
| chc ($\langle behaviour \rangle$,$\langle behaviour \rangle$)
| rec $\langle behaviourVariable \rangle$ ($\langle behaviour \rangle$)
| spn ($\langle behaviour \rangle$)
| psh ($\langle lable \rangle$,$\langle sessionType \rangle$)
| $\langle region \rangle$ ! $\langle behaviourVariable \rangle$
| $\langle region \rangle$ ? $\langle behaviourVariable \rangle$
| $\langle region \rangle$ ! $\langle region \rangle$
| $\langle region \rangle$ ? $\langle lable \rangle$
| $\langle region \rangle$ ! $\langle lable \rangle$
| $\langle region \rangle$ ? optn[$\langle oplist \rangle$]

$\langle oplist \rangle$ ::= ( $\langle lable \rangle$ ; $\langle behavioiur \rangle$ ) $\langle opt\_feild \rangle$

$\langle opt\_feild \rangle$ ::= , ( $\langle lable \rangle$ ; $\langle behavioiur \rangle$ )
| $\epsilon$

### Constraints

Constraints are similar to behaviours but also make use of the sub-grammar for bTypes, regions and session types which are given below.

$\langle \mathit{constr} \rangle$ ::= $\langle \mathit{bType} \rangle$ < $\langle \mathit{bType} \rangle$
  | $\langle \mathit{behaviour} \rangle$ < $\langle \mathit{behaviour} \rangle$
  | $\langle \mathit{region} \rangle \sim \langle \mathit{regionVar} \rangle$
  | $\langle \mathit{channel} \rangle \sim \langle \mathit{sessionType} \rangle$
  | $\langle \mathit{channelEnd} \rangle \sim \langle \mathit{sessionType} \rangle$
  | $\langle \mathit{contr} \rangle$ , $\langle \mathit{constr} \rangle$
  | $\epsilon$

### Types

The implementation of types is relatively simple. They can be any of the forms listed below and use the sub-grammar for regions.

$\langle \mathit{bType} \rangle$ ::= unit
  | bool
  | int
  | pair ( $\langle \mathit{bType} \rangle$ ; $\langle \mathit{bType} \rangle$ )
  | funct $\langle \mathit{bType} \rangle$ -> $\langle \mathit{bType} \rangle$ - $\langle \mathit{behaviourVariable} \rangle$
  | ses $\langle \mathit{region} \rangle$
  | $\langle \mathit{TVar} \rangle$

### Region variables

Regions are very simple and either consist of a region variable or a label.

$\langle \mathit{regionVar} \rangle$ ::= $\langle \mathit{lable} \rangle$
  | $\langle \mathit{region} \rangle$

### Session types

Session types follow a similar pattern to behaviours.

$\langle sessionType \rangle$ ::= end
| ! $\langle bType \rangle$ $\langle sessionType \rangle$
| ? $\langle bType \rangle$ $\langle sessionType \rangle$
| ! $\langle sessionType \rangle$ $\langle sessionType \rangle$
| ? $\langle sessionType \rangle$ $\langle sessionType \rangle$
| (+) [$\langle sesOpL \rangle$] ($\langle lable \rangle$ ; $\langle sessionType \rangle$)
| + [$\langle sesOpL \rangle$] [$\langle sesOpL \rangle$]
| $\langle SVar \rangle$

$\langle sesOpL \rangle$ ::= ($\langle lable \rangle$ ; $\langle sessionType \rangle$) $\langle ses\_opt\_field \rangle$

$\langle ses\_opt\_field \rangle$ := , ($\langle lable \rangle$ ; $\langle sessionType \rangle$)
| $\epsilon$

## 3.3 OCaml Types

In order for the parser to have the correct types for dealing with behaviours and constraints types for these first needed to exist. This involved creating new types in OCaml for each of Behaviours, Constraints, Types, Regions and Session types. These main types are built up of subtypes, strings and, in the case of 'Tau' and 'End' nothing.

As an example the type for behaviours takes the following form:

```
type b =
  | BVar of string
  | Tau
  | Seq of seq
  | ChoiceB of choiceB
  | RecB of recB
  | Spawn of spawn
  | Push of push
  | SndType of outT
  | RecType of recT
  | SndReg of sndR
  | RecLab of recL
  | SndChc of sndC
```

```
  | RecChoice of recC
  (* None included due to requirements to run main file *)
  | None
and sndC = { regCa : string ; labl : string}
and recC = { regCb : string ; cList : (string * b) list }
and recL = { regL : string ; label : string}
and sndR = { reg1 : string ; reg2 : string}
and recT = { regionR : string ; outTypeR : t}
and outT = { regionS : string ; outTypeS : t}
and push = { toPush : stackFrame}
and spawn = { spawned : b}
and recB = { behaVar : string ; behaviour : b}
and choiceB = {opt1 : b ; opt2 : b}
and seq = {b1 : b ; b2 : b};;
```

It can be seen here how RecChoice, the external choice type, is made up of the subtype recC which in turn consists of a string and list of (string, behaviour) tuples. The other types are implemented in a similar way.

As well as implementing the types each type also has a to_string function. This is to allow for the re-printing of the input code as part of the output.

### 3.3.1 Challenges

Again the main challenges with implementing this part of the project was the new language. While OCaml has excellent documentation it was not easy to find examples or tutorials to help with implementing such an interlinked system of types. One tutorial that was helpful was [11]. The book Types and Programming Languages [12] was also helpful.

## 3.4 Behaviour Checker

The behaviour checker implements the rules from (fig. 2.4). The first step towards implementing these was to store the relevant information in an easily accessible form. In this case that meant storing the relevant constraints. Then the behaviours can be checked and the relevant actions from the rules applied.

### 3.4.1 Storing the constraints

The constraints relevant to the Behaviour Checker are the region constraints, the behaviour constraints and the type constraints. Each is stored in a slightly different way to account for the fact that the format and usage of each of the constraint types is different. Other constraints are irrelevant to the behaviour checker and so are not stored.

#### Behaviour constraints

These constraints are of the form

$$b \subseteq \beta$$

where b is a behaviour and $\beta$ is a behaviour variable. Behaviour constraints are used in the Beta and Rec rules. In the Beta rule ($\Delta \models \beta \rightarrow c\Delta \models b$ if $C \vdash b \subseteq \beta$) the constraints are used to replace the behaviour variable beta with each behaviour associated with it. Each of these is then checked against the rules to see if it produces a valid end state consisting of the empty stack and the $\tau$ behaviour.

In the recursion (Rec) rule the behaviour constraints are used where there are any constraints of which the left hand side matches the current recursion behaviour. If any are found then the left hand side of that rule is replaced with $\tau$ in the call to check the body of the recursion.

The decision was made to store the behaviour constraints as a hash table. The key is the right hand side of the constraint ($\beta$) and the value is the left hand side (b, the behaviour associated with $\beta$). In this way, in the case of either rule, we can search quickly and find all values associated with the key and have them returned as a list. We can then either replace $\beta$ with each value from the list in turn (Beta Rule) or we can search the list to find and replace any instances of the current recursive behaviour (Rec Rule).

#### Functional type constraints

Type constraints are of the form

$$T_1 <: T_2$$

and are used in the Out Rule as well as in endpoint relation checks in the Delegate Rule. They are used according to semantics given in (fig. 3.1) that state that if a constraint stating the fact exists then $T_1$ is a functional subtype of $T_2$. If both $T_1$ and $T_2$ are pairs then if the first type of the first pair is a subtype of the first type of the second pair and if the same is true of the second types of both pairs then the first pair is a subtype of the second. It is a similar case for functional types.

If both types are of the form $Ses^{\rho}$ then if there exists a region constraint linking the $\rho$'s they are functional subtypes.

Type constraints are stored as a hash table with the right hand type as the key and the left hand type as the value. Since type constraints are also transitive this means that when checking these constraints first the right hand side of the constraint to be checked is searched for. A list of associated subtypes for this type are returned. Each of these is then checked to see if it matches the left hand side. If not the search is repeated with this new type as the right hand side. This continues until either the left hand side is found or there are no more types to check.

Since type constraints are also reflexive it is always checked if the two types are equal first.

Type constraints are also used for session sub-typing (fig. 3.2). They are used in this situation when checking if two session types are either outputting confined types or inputting them where the types must be functional subtypes in order for the session types to be related. Checks on session sub-types are performed in the delegation rule.

Other session types that need to be checked to see if they are subtypes are internal and external choice types. For internal choice one choice is a subtype of another choice if the number of options in the list in the sub-type is less than or equal to the number of options in the super-type and if for all options in the second list the corresponding session type in the first list is a subtype of the session type.

For external choice one choice is a sub-type of another if the first list (values that the choice must be able to accept) of the sub-type is a subset of the first list of the super-type and if the union of the lists of the super-type is a subset of the union of the lists of the sub-type. As well as this for all elements of either list of the super-type the corresponding session type of the sub-type must itself be a sub-type of the session type of that element.

$$\frac{(T_1 \subseteq T_2) \in C}{C \vdash T_1 <: T_2} \quad \frac{C \vdash \rho \sim \rho'}{C \vdash \mathsf{Ses}^{\rho} <: \mathsf{Ses}^{\rho'}} \quad \frac{C \vdash T_1' <: T_1 \quad C \vdash \beta \subseteq \beta' \quad C \vdash T_2 <: T_2'}{C \vdash T_1 \xrightarrow{\beta} T_2 <: T_1' \xrightarrow{\beta'} T_2'}$$

$$\frac{}{C \vdash T <: T} \quad \frac{C \vdash C \vdash T_1 <: T_2 \quad C \vdash T_2 <: T_3}{C \vdash T_1 <: T_3} \quad \frac{C \vdash C \vdash T_1 <: T_2 \quad C \vdash T_3 <: T_4}{C \vdash T_1 \times T_3 <: T_2 \times T_4}$$

Figure 3.1: Functional Sub-typing

$$\frac{C \vdash T_2 <: T_1 \quad C \vdash \eta_1 <: \eta_2}{C \vdash !T_1.\eta_1 <: !T_2.\eta_2} \quad \frac{C \vdash T_1 <: T_2 \quad C \vdash \eta_1 <: \eta_2}{C \vdash ?T_1.\eta_1 <: ?T_2.\eta_2}$$

$$\frac{m \leq n \quad \forall i \leq n. C \vdash \eta_i <: \eta_j}{C \vdash \oplus \underset{i \in [1,m]}{\{L_i : \eta_i\}} <: \oplus \underset{j \in [1,n]}{\{L_j : \eta_j\}}} \quad \frac{I_1 \subseteq J_1 \quad J_1 \cup J_2 \subseteq I_1 \cup I_2 \quad \forall (i \in J_1 \cup J_2). C \vdash \eta_i <: \eta_i'}{C \vdash \& \underset{i \in (I_1, I_2)}{\{L_i : \eta_i\}} <: \& \underset{i \in (J_1, J_2)}{\{L_i : \eta_i'\}}}$$

Figure 3.2: Session Sub-typing

**Region constraints**

Region constraints take the form of either

$$\rho \sim \rho' \text{ or } \rho \sim l$$

and are used in the rules: Out, In, Del, Res, ICh and ECh. They are used in the same way for each of the rules, which is, to check if there is a link between a region and a label. The complication arises from the fact that regions can be chained and then linked to a label. For example the constraints $\rho_1 \sim \rho_2$, $\rho_2 \sim \rho_3$ and $\rho_3 \sim l1$ tells us that $\rho_1, \rho_2$ and $\rho_3$ are all linked to $l1$.

Due to this region constraints are stored as a list of tuples where each tuple consists of a label and a hash table with region variables as keys and units as the values. In this way when looking up a particular label and region we can simply search for the label in the list and then check the hash table for the region. This is an efficient implementation since the number of labels in any given program is unlikely to be excessive.

Storing the constraints in this way, in a single pass through the constraint list, was challenging. It was achieved by first creating a new tuple for the next constraint to be stored. This consists of either a label or a 'None' value and a hash table with either one or two regions depending on the structure of the constraint. All other elements from the list that match either of the two values associated with the new constraint are then found and removed from the list. They are merge together with the new element to form a single list element which is then added back to the list.

### 3.4.2 Behaviour checker function

The behaviour checker verifies that the input program will in fact communicate correctly. It takes an input of the behaviours produced from the first level (2.2) and the associated constraints stored in the method described above. The behaviours are then checked against the rules from (fig. 2.4) and the appropriate actions taken.

The parameters to the function are a set of constraints and a list of behaviour tuples of the form (*behaviour*, *stack*, *stack_labels*, *continuation*). The stack represents the one described in the rules and the stack labels structure is used to ensure that each label is only ever pushed to the stack once. The continuation is used to keep track of

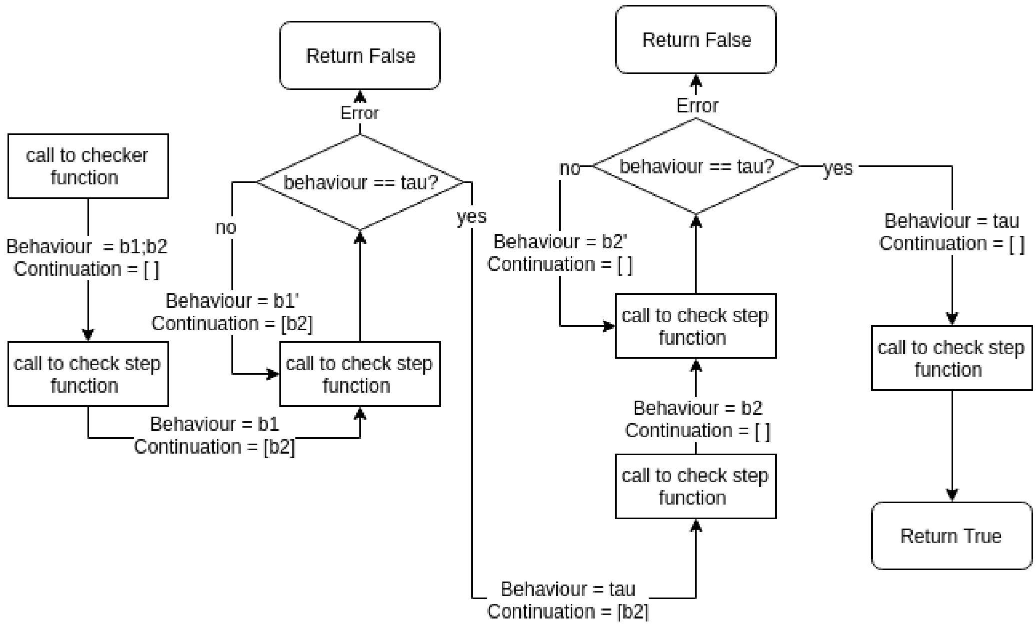


Figure 3.3: Application of Sequence Rule

any behaviours that need to be dealt with once the current behaviour is finished with, see (fig. 3.3) for an example.

The initial function called is a wrapper function that calls to a check step function with the first tuple from the list. The check_step function then takes the current stack and checks to see if the top frame contains the 'End' session type (i.e. it checks if the end rule applies) if it does it removes this frame and then it continues to check to see which, if any, of the rest of the rules apply. The checks are performed in this order since the End Rule is performed based on the state of the stack while all other rules are performed based on the current behaviour as well as the state of the stack.

### 3.4.3 Results

The behaviour checker will output the results of the check. In the case of the check failing it will also print the name of the rule on which the check failed and if it failed due to a failed constraint check. It will not print the location of the error in the source code since this was outside the scope of this project.

If the check is successful it has shown that the input code follows the correct protocol for the communication channel. It has also shown partial lock freedom due to the well stackness of the program.

### 3.4.4 Challenges

The main technical challenge encountered in this implementation of the checker was, again, the new language. However this was overcome more quickly when developing this section of the project due to the experience gained with the language in implementing the previous sections of the project.

Other technical difficulties that were encountered included dealing with the different implementations of stack and hash tables in the standard library and the core_kernal library. The stacks in the core_kernel library [13] included functions (such as exists and top) that made for a nicer implementation but the hash tables were missing the functionality to return lists of all bindings to a key which was necessary for the constraint checks. This was overcome when I realised it was possible to rename the hash table module from the standard library before importing the core_kernel library and so avoid the shadowing of the binding.

The implementation of the region constraint storage also took quite some time since it is rather complicated code and pushed my knowledge of the OCaml language.

# Chapter 4

# System Examples

In this chapter some sample programs are shown and the process through which they are run through both levels of the system is explained.

## 4.1 Simple Swap Service

The examples given in the paper [1] include an example for a simple swap service which is also discussed briefly in the introduction to this report. It consists of two clients and a coordinator. The clients send values to the coordinator who then returns to client one the value sent by client two and vice versa.

### 4.1.1 Input code

```
let fun coord(_) =
  let val p1 = acc swp ()
      val x1 = recv p1
      val p2 = acc swp ()
      val x2 = recv p2
  in send p2 x1; send p1 x2;
     coord ()
in spawn coord;
```

```
let fun swap(x) =
  let val p = req swp ()
  in send p x; recv p
in spawn (fn _ => swap 1);
   spawn (fn _ => swap 2);
```

In this code the coordinator accepts two connects on the channel swp using acc-swp (). This gives it two endpoints, p1 and p2. After accepting each connection it receives a value over that connection. Finally the received values are sent over the alternate endpoint.

In the swap processes a connection is requested on the swp channel using req-swp (), giving the endpoint p. A value, in this case either 1 or 2, is then sent over this endpoint and finally some other value is received over p.

The endpoints used by the coordinator, received from calling acc-swp (), are used according the session type $?T.!T.end$. This means that on this connection a value of type $T$ is received, then a value of the same type is sent then the connection ends.

The endpoint used by the swap function, received from the call to req-swp (), is used according to the session type $!Int.?T'.end$. In this case an int is sent, a value of some type $T'$ is received and then the connection is terminated.

From examining these session types and the above code it is clear that types $T$ and $T'$ must be the same, due to the fact that the values sent by the coordinator are the ones that are then received by the swap functions. Furthermore both of these types must be of the type *int* since the values received by the coordinator are sent by the swap functions and these values are integers (1 and 2).

In this case, the programs can communicate and are typable when $T = T' = Int$. The inference algorithms proposed in (Spacasassi & Koutavas)[1] can infer these session types from the code given.

### 4.1.2 Intermediate code

The first level of the system will produce code detailing the behaviour of the given input code and the constraints relevant to it. A simplified version of this can be seen here and the full version in the appendix.

```
Behaviours:
spn(B101);
spn(psh($I3$, S126);R131 ! int;R132 ? T125);
spn(psh($I3$, S126);R147 ! int;R148 ? T141)

Constraints:
Cswap1' ~ S105,
Cswap2' ~ S106,
Cswap3 ~ S126,
```

```
rec B101(psh($l1$, S105);R114 ? T103 ;psh($l2$, S106);R115 ? T104;
    R116 ! T103;R117 ! T104;B101) < B101,
$l1$ ~ R114,
$l1$ ~ R117,
$l2$ ~ R115,
$l2$ ~ R116,
$l3$ ~ R131,
$l3$ ~ R132,
$l3$ ~ R147,
$l3$ ~ R148
```

This code is given in the syntax developed by me for this project (see 3.1). In this case the first spawn of behaviour variable B101 is referencing the spawning of the coordinator in the input code. Behaviour variable B101 then references the behaviour of the coordinator. The remaining two spawns reference the spawns of the swap functions.

In the constraints we can first see the three channel constraints. These are referring to the endpoints returned from the calls to acc-swp and req-swp. The next constraint is a behaviour constraint. This is what gives us the link between the behaviour variable B101 and the actual behaviour of the coordinator. Finally the last constraints are the region constraints. These either link regions to other regions or labels to regions. The labels here are static approximations of the locations of the endpoints in the code. In this case $l1$ refers to the location in the coordinator function where acc-swp is called to generate the connection for endpoint p1. $l2$ references the location of p2 and $l3$ references the call to req-swp in the spawn function.

Since the behaviour checker implementation does not deal with inference we must manually apply some substitutions before we can run this code through the it. This involves replacing the session variables with the actual session endpoint types and the type variables with the actual types. These substitutions are detailed in table 4.1.

### 4.1.3 Behaviour check

The intermediate code with substitutions can then be run through the behaviour checker detailed in (sec. 3.4). Here the steps which would be taken to check the simplified code are outlined.

| | |
|---|---|
| S105<br>S106 | ?int !int end |
| S126 | !int ?int end |
| T125<br>T141<br>T103<br>T104 | int |

Table 4.1: Manual Substitutions for Simple Swap

First the code is run through the parser and lexical analyser. The resulting constraints are then passed into the function that deals with their internal storage (see sec. 3.4.1). The simplified output from running this example, which includes the printing of the internal storage of the constraints, can be seen below. It can clearly be seen here how the internal storage allows for efficient look ups, particularly with respect to region constraints. The stored constraints and the behaviours are then passed to the behaviour checker.

The first behaviour is a sequence. Once the rule is applied to deal with this the next behaviour to look at is the $spn(B101)$. At this stage the stack is empty. This behaviour matches the rule SPAWN from (fig. 2.4). This then calls the check function again with the behaviour tau and the result is anded with the result of a call to the check functions where the behaviour is the body of the spawn ($B101$).

The rules then continue to be applied in this fashion until a state is reached where the stack is empty, the continuation is empty and the behaviour is tau. In this case the check function returns true. Alternately if a state other than this is reached where no rule can be applied false is returned along with a message detailing in which rule the error state was reached. A detailed diagram of how the simplified code would be dealt with in the behaviour checker can be found in (fig. 4.1).

```
Behaviours:
Spn(B101);
Spn(Psh(l3, ! int ? int end ); R131 ! int; R132 ? int);
Spn(Psh(l3, ! int ? int end ); R147 ! int; R148 ? int)

Paired:
  Beta: B101
```

```
  Behaviour: rec B101 (Psh(l1 , ? int ! int end ); R114 ? int; Psh(
     l2 , ? int ! int end ); R115 ? int; R116 ! int; R117 ! int;
     B101)

Region Constraints:
 label: l3
 regions:
  R132, R148, R131, R147
label: l2
 regions:
  R116, R115
label: l1
 regions:
  R117, R114

check successful!
```

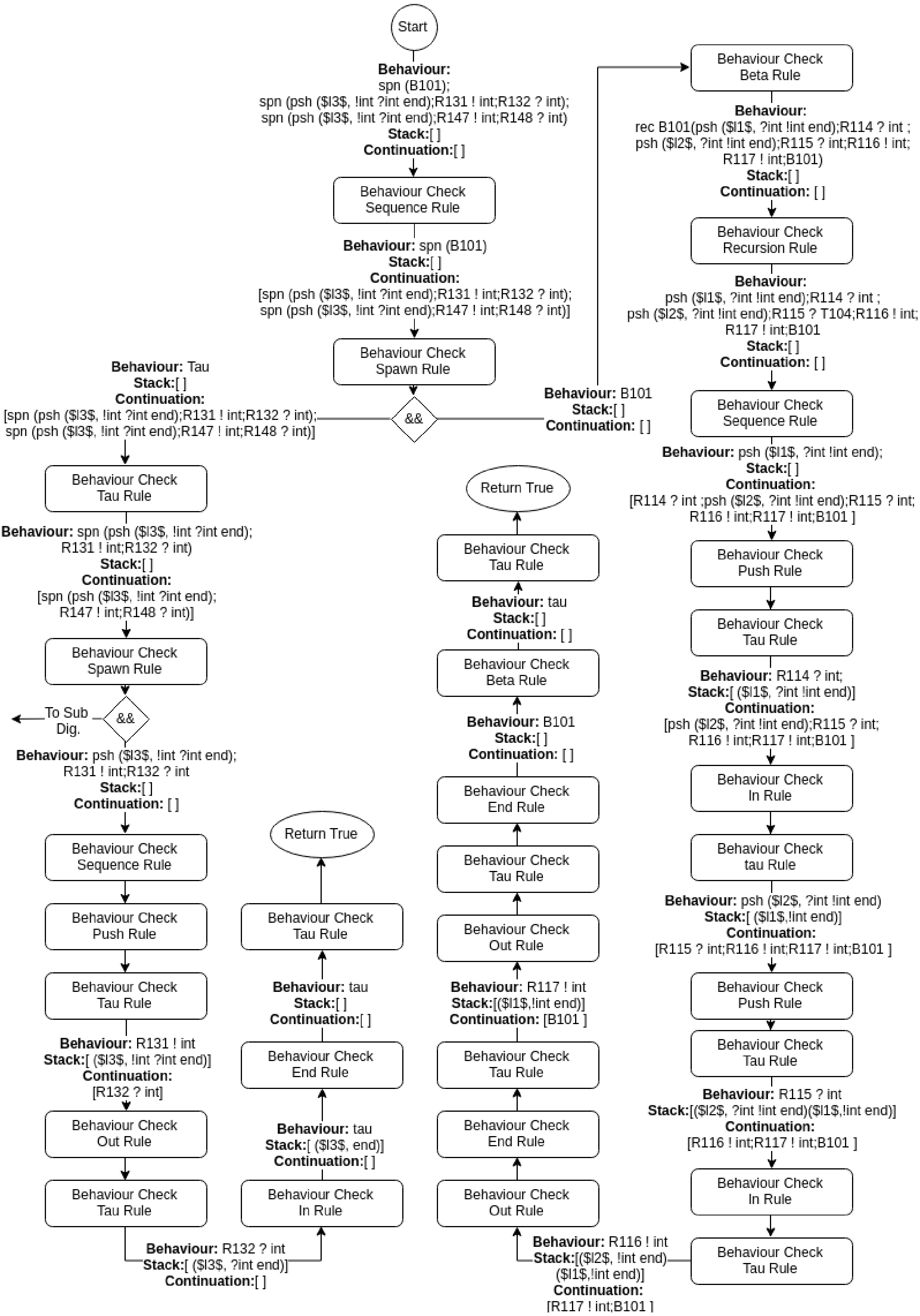


Figure 4.1: Simple Swap Path Taken Through Behaviour Check, Main Dig.

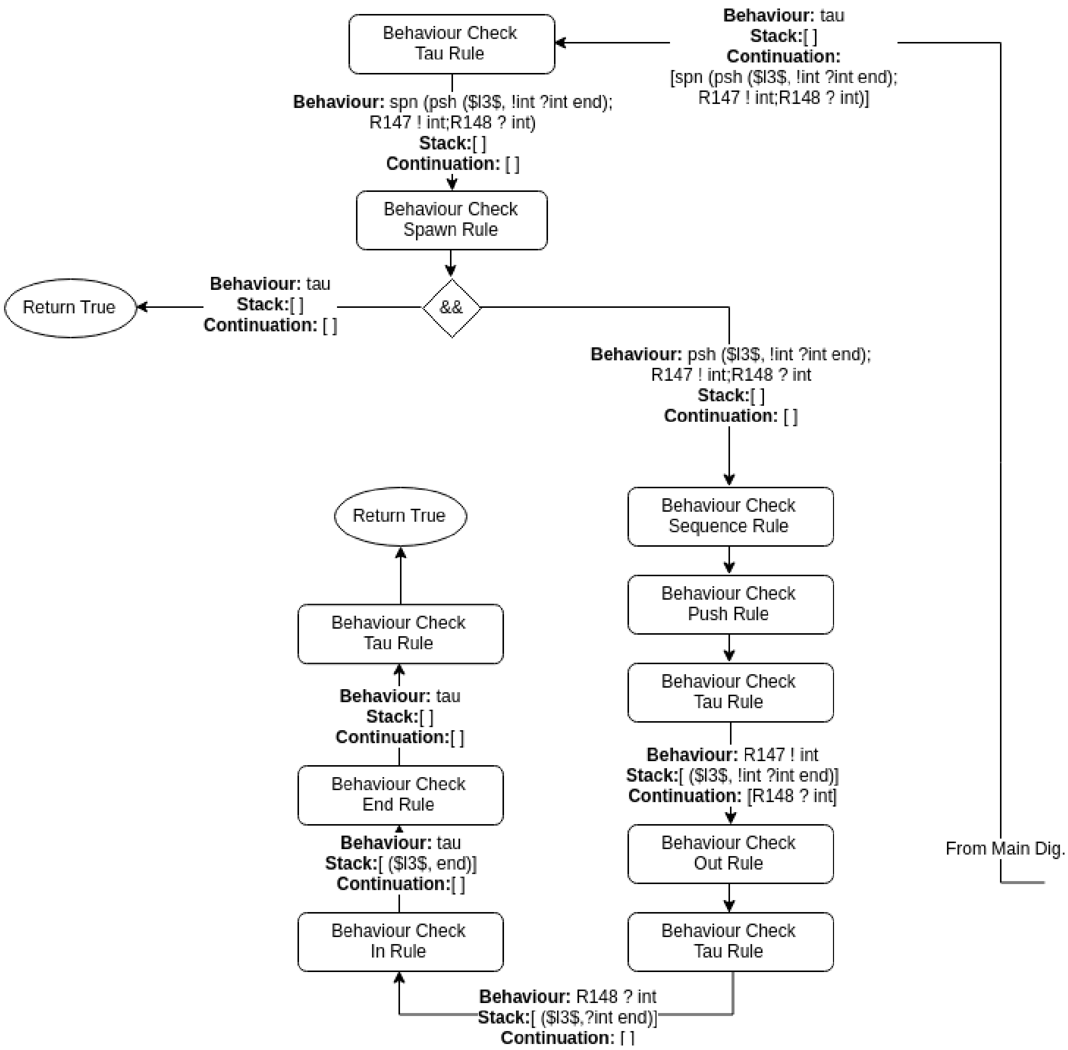


Figure 4.2: Simple Swap Path Taken Through Behaviour Check, Sub Dig.

## 4.2 Swap Delegation

This example is based on one that is given in the paper (Spacasassi & Koutavas)[1]. Here it has been altered and extended to show how incorrect code would be dealt with by the behaviour checker.

In this example the coordinator function delegates the exchange to the clients. In the previous example the coordinator function acts as a bottleneck when there are large exchange values, the delegation in this example avoids this.

### 4.2.1 Input code

Here the swap function connects to the coordinator using req-swp as before but now it offers two choices; SWAP and LEAD. If SWAP is selected by the coordinator then the function receives on the endpoint p then sends the value x over p.

Otherwise if LEAD is selected the function resumes the endpoint q, which is received over p. A value is the received on q and the value passed in to the function is then sent over q.

The coordinator accepts two connections on the swp channel and gets the two endpoints p1 and p2. It then selects SWAP on p1 and LEAD on p2. Then p1 is sent over p2 and the function recurs.

```
let fun coord(_) =
  let val p1 = acc swp ()
  in sel SWAP p1;
    let val p2 = acc swp
    in sel LEAD p2;
       deleg p2 p1;
       coord()
```

```
let fun swap(x) =
  let val p = req swp ()
  in case p {
    SWAP: recv p; send p x
    LEAD: let val q = resume
       p
               val y = recv q
             in send q x; y }
```

When this coordinator function is analysed in isolation the inference algorithms will infer that the endpoints p1 and p2 have the type $\eta_{i,i\in(p1,p2)} = (\oplus\{(SWAP : \eta'), (LEAD : \eta'.end)\})$. When swap is analysed they will infer that $\eta_p = \Sigma\{(SWAP : ?T_1.!T_2.end), (LEAD : ?\eta_q.end)\}$ and $\eta_q = ?T_2.!T_1.end$.

When the coordinator is looked at in isolation then $\eta'$ can be any endpoint. However due to duality $\eta'$ must be equal to $\eta_q$ in this case. Also $T_1 = T_2$ must be true.

The deliberate error in this code is that in the swap function when SWAP is selected a value is received then a value is sent. When LEAD is selected the same pattern is followed. This will produce a deadlock where both instances of swap are waiting on the other to send a value.

### 4.2.2 Intermediate code

A simplified version of the output from the first level of the system is given here.

```
Behaviours:
spn(B118);
spn(psh($I3$, S142);R150? optn [($SWAP$ ; R152 ! int;R151 ? T160),
    ($LEAD$ ; R153 ? $I4$;R154 ? T157;R155 ! int)]);
spn(psh($I3$, S142);R181? optn [($SWAP$ ; R183 ! int;R182 ? T191),
    ($LEAD$ ; R184 ? $I4$ ;R185 ? T188;R186 ! int)])

Constraints:
unit <  ses R151,
unit <  ses R182,
Cswap1' ~ S120,
Cswap2' ~ S121,
Cswap3 ~ S142,
rec B118(psh ($I1$, S120);R128 ! $SWAP$;psh ($I2$, S121);R129 !
   $LEAD$;R130 ! R131;B118) < B118,
$I1$ ~ R128,
$I1$ ~ R131,
$I2$ ~ R129,
$I2$ ~ R130,
$I3$ ~ R150,
$I3$ ~ R152,
$I3$ ~ R153,
$I3$ ~ R181,
$I3$ ~ R183,
$I3$ ~ R184,
$I4$ ~ R154,
$I4$ ~ R155,
```

| | |
|---|---|
| S120<br>S121 | $(+)[(\text{SWAP}; ?int!intend), (\text{LEAD}; !?int!intendend)]$ |
| S142 | $+[(\text{SWAP}; ?int!intend), (\text{LEAD}; ??int!intendend)][\,]$ |
| T160<br>T191<br>T157<br>T188 | int |

Table 4.2: Manual Substitutions for Delegated Swap

```
$I4$ ~ R185,
$I4$ ~ R186
```

Again substitutions must be made before this code can be given to the behaviour checker. These are detailed in table 4.2.

### 4.2.3 Behaviour check

As you can see from the code given below the behaviour checker has recognised that the input code cannot communicate correctly. The error given is 'out rule stack frame incorrect for current behaviour'. This tells us that the checker ran into problems when it was trying to verify a behaviour of the type $\rho!T$. We also know, from looking up the rules, that the stack frame did not match the expected form $(l; !T\eta)$. From these we can deduce that the checker ran into the error when attempting to check the swap option against the current stack frame since the option is attempting to send a value while the stack frame is expecting it to attempt to receive one.

```
Behaviours:
    Spn(B118);
    Spn(Psh(I3, +[(SWAP;? int ? int end ),(LEAD;? ? int ! int end
        end)][]); R150 ? optn [(SWAP; R152 ! int ; R151 ? int)(LEAD;
         R153 ? I4; R154 ? int; R155 ! int)]);
    Spn( Psh( I3, +[(SWAP;? int ? int end ),(LEAD;? ? int ! int end
         end)][]  ); R181 ? optn [(SWAP; R183 ! int; R182 ? int)(
        LEAD;  R184 ? I4 ; R185 ? int; R186 ! int)])
Behaviour constraints:
Beta: B118
```

```
Behaviour: rec B118 (Psh(l1, (+)[ (SWAP;? int ! int end ), (LEAD;!
    ? int ! int end end)]) ; R128 ! SWAP; Psh(l2, (+)[ (SWAP;? int
    ! int end ), (LEAD;! ? int ! int end end )]); R129 ! LEAD;
   R130 ! R131;B118)

Region Constraints:
 label: l4
 regions:
R154, R185, R186, R155
label: l3
 regions:
R152, R183, R184, R181, R150, R153
label: l2
 regions:
R129, R130
label: l1
 regions:
R128, R131

Type Constraints:
Paired:
Super: ses R182
sub:  unit
Paired:
Super: ses R151
sub:  unit

out rule
stack frame incorrect for current behaviour
ERROR: Failed Check
```

## 4.3 TLS

A very simple implementation of the Transport Layer Security (TLS) handshake has been detailed here to show how it would operate under this system. Again the simplified version of the output from the two layers has been shown here and the full version has been included in the appendix.

### 4.3.1 Input code

Here we see a very simple version of the TLS handshake implemented in the version of $ML_s$ accepted by the first level. In this implementation we have a client and a server. The client establishes a connection on the tls channel and sends a number, this represents the client hello message. It then receives four values on this connection. These represent the server hello, server certificate, server exchange key and the server hello done messages. The client finally sends the client exchange key and the change cipher spec. The communication ends when the client receives the value representing the change cipher spec message from the server.

The communications on the server side are the complements of those made by the client. Since all values are represented as integers we can conclude that the session types of these endpoints should be $!int.?int.?int.?int.?int.!int.!int.?int$ and $?int.!int.!int.!int.!int.?int.?int.!int$ for the client and server respectively.

```
let client = fun z =>
  (let p1 = req tsl () in
    ((send (p1, 1));
     (recv p1);
     (recv p1);
     (recv p1);
     (recv p1);
     (send (p1, 1));
     (send (p1, 2));
     (recv p1)
     )) in
```

```
let server = fun z =>
  ( let p2 = acc tsl () in
   (recv p2);
    (send (p2, 3));
    (send (p2, 3));
    (send (p2, 3));
    (send (p2, 3));
    (recv p2);
    (recv p2);
    (send (p2, 3))
      )
  in (spawn (client) );
(spawn (server))
```

### 4.3.2 Intermediate code

Here we have taken the output from the first level and simplified it. Again the substitutions detailed in table 4.3 must be made before this code can be run through the behaviour checker.

```
Behaviours:
spn(psh($I1$, S190);R201 ! int;R202 ? T185;R203 ? T186;R204 ? T187
    ;R205 ? T188;R206 ! int;R207 ! int;R208 ? T209);
spn(psh($I2$, S216);R227 ? T213;R228 ! int;R229 ! int;R230 ! int;
   R231 ! int;R232 ? T214;R233 ? T215;R234 ! int)

Constraints:
CtsI1 ~ S190,
CtsI2' ~ S216,
$I1$ ~ R201,
$I1$ ~ R202,
$I1$ ~ R203,
$I1$ ~ R204,
$I1$ ~ R205,
$I1$ ~ R206,
$I1$ ~ R207,
$I1$ ~ R208,
$I2$ ~ R227,
$I2$ ~ R228,
$I2$ ~ R229,
$I2$ ~ R230,
$I2$ ~ R231,
$I2$ ~ R232,
$I2$ ~ R233,
$I2$ ~ R234
```

### 4.3.3 Behaviour check

As you can see from the output below the behaviour checker shows that this communication is valid when the given substitutions are made.

| | |
|---|---|
| S190 | $!int.?int.?int.?int.?int.!int.!int.?int$ |
| S216 | $?int.!int.!int.!int.!int.?int.?int.!int$ |
| T185 | |
| T186 | |
| T187 | |
| T188 | |
| T209 | int |
| T213 | |
| T214 | |
| T215 | |

Table 4.3: TLS Substitutions

```
Behaviours:
Spn(Psh(l1, ! int ? int ? int ? int ? int ! int ! int ? int end);
   R201 ! int; R202 ? int; R203 ? int; R204 ? int; R205 ? int;
   R206 ! int; R207 ! int; R208 ? int);
Spn(Psh(l2, ? int ! int ! int ! int ! int ? int ? int ! int end);
   R227 ? int; R228 ! int; R229 ! int; R230 ! int; R231 ! int;
   R232 ? int; R233 ? int; R234 ! int)

Behaviour constraints:

Region Constraints:
 label: l2
 regions:
R228,R230,R231,R229,R234,R232,R227,R233
label: l1
 regions:
R205,R201,R204,R207,R202,R203,R206,R208

Type Constraints:


check successful!
```

The code given here is the simplified version.

# Chapter 5

# Evaluation

This chapter explains how the application was evaluated as well as giving a reflection on the results of the evaluation.

## 5.1 Testing

Testing was conducted in stages. Firstly the parser and lexer were tested, then the constraint storage and finally the behaviour checker.

### 5.1.1 Lexer and parser

The testing for the lexer and parser was relatively simple, to_string methods were written for each of the data types and constraint data structures and the resulting behaviours and constraints from the parsing were printed out to the console.

Initially they were printed in the exact form that they were input. This allowed a side by side comparison of the input and output which then allowed for errors to be spotted quickly and easily.

### 5.1.2 Constraint storage

Testing for the constraint storage was done in a similar way to that of the lexer and parser. Instead of printing the constraints exactly as they were read in the function was updated to print them as they were been stored. This allowed for them to be quickly checked against the input to see if they were been stored correctly.

For example in the case of region constraints the input could contain constraints such as:

```
R12 ~ $l1$ ,
R11 ~ R12,
R15 ~ $l3$
```

Which should then be output in the form:

```
label: l1
regions: R11, R12
label: l2
regions: R15
```

It is then easy to check if the regions listed under a particular label are in fact linked to it.

### 5.1.3 Behaviour checker

The testing for the behaviour checker was the most in depth. A test suit has been developed that includes the examples given in 4. As well as these over 20 small programs were written to test the individual rules one at a time.

These small programs were written to test each of the rules implemented for the behaviour checker (fig. 2.4). A series of behaviours were developed to simulate a simple situation where each rule would pass i.e. where the stack frame and behaviour matched the rules requirements and all constraints were met. Test were then written where each of the conditions for the rule to pass were broken in turn in order to show that the behaviour checker would fail as expected.

Since testing that the rules fail correctly involved testing situations where the constraint checks fail these tests also show that the constraint storage and look-up functions work as they should.

## 5.2 TLS Check

When writing the input code for the TLS example (sec. 4.3) the initial output from the first level compiler that had been implemented for the paper [1] included a constraint stating that the final receive performed by the client should be receiving an value of type unit. However there was also a constraint stating that the corresponding send from the server should be sending a value of type int.

It was discovered then that the inference rule for spawn was outdated. It would always infer that the if the final operation in a spawned function was to receive then the final value was of the unit type. This was producing a false negative result from the system. This has been updated so that the system can now allow this type of program to be typed correctly.

## 5.3 Reflection

The test detailed in the previous section have shown that all components of the program behave as expected. The development of the project in a functional style meant that once the files compiled correctly the behaviour of the program was almost always what was specified. In the main the problems discovered by the tests were misunderstandings of the rules or the constraints. The tests made these misunderstandings obvious and so they could be found and fixed.

The test was also invaluable in showing where the error hints from the behaviour checker should display. For example the tests for the ICh rule showed an uncaught Not_found exception instead of displaying an error message. This allowed me to go back to the program, catch the exception and output a more helpful message.

# Chapter 6

# Conclusion

The objectives of this project were to first develop an understanding of Session types and behaviours specifically related to the work done in (Spacasassi & Koutavas)[1] and to implement the design for a behaviour checker detailed in said paper. This has been done in the fashion detailed in this report. The early chapters of this report detail the depth of my understanding of session types and behaviours as well as my understanding of the paper [1]. The implementation of the behaviour checker is complete and is detailed in chapter 3, examples of its use have been detailed in chapter 4, and its testing has been described in chapter 5.

## 6.1 Evaluation

In the end I feel that the correct choice was made in terms of the language and tools used in this project. Though Menhir and OCamlLex were the second choice for the creation of the parser they were far more suited to the project then the initial choice, Camlp4, would have been.

## 6.2 Future Developments

Future improvements to this project could include extending the behaviour checker implemented to include the inference algorithms. As well as this a true implementation of error messages could be added.

## 6.3 Achievements

Over the course of this project I have increased my knowledge of session types and of how this relatively new technology works. Since this technology is currently at a research stage it is not something I had come across previously and so was both interesting and challenging to learn about.

As well as gaining the skills to write complex code in OCaml a language with which I was not previously familiar. I have also learned to use OCamlLex and Menhir to create a parser.

The testing of my implementation of the behaviour checker also uncovered a weakness in the previously implemented first level of the system, which is detailed in section 5.2. Feedback on this, when given to the authors of the paper [1], allowed them to make the type system more programmer friendly by reducing the number of programs that would produce false negatives in the system.

# Code Repository

The source code and tests for this project can be found at `https://github.com/taj121/FVST`.

# Appendix

## Simple Swap

### Full intermediate code

The substitutions to run this code through the behaviour checker are given in table 4.1.

```
tau;tau;tau;spn (B101);tau;spn (B118);tau;spn (B134)

Cswap1' ~ S105,
Cswap2' ~ S106,
Cswap3 ~ S126,
tau < B41,
tau < B57,
tau < B85,
psh ($I1$, S105) < B108,
psh ($I2$, S106) < B110,
psh ($I3$, S126) < B128,
psh ($I3$, S126) < B144,
R116 ! T103 < B112,
R117 ! T104 < B113,
R131 ! int < B129,
R147 ! int < B145,
R114 ? T103 < B109,
R115 ? T104 < B111,
R132 ? T125 < B130,
R148 ? T141 < B146,
```

```
tau;tau;B123;tau < B118,
tau;tau;B128;tau;tau;tau;B129;tau;tau;B130 < B123,
tau;tau;B139;tau < B134,
tau;tau;B144;tau;tau;tau;B145;tau;tau;B146 < B139,
rec B101(tau;tau;B108;tau;tau;B109;tau;tau;B110;tau;tau;B111;tau;
   tau;tau;B112;tau;tau;tau;B113;tau;tau;B101) < B101,
$I1$ ~ R114,
$I1$ ~ R117,
$I2$ ~ R115,
$I2$ ~ R116,
$I3$ ~ R131,
$I3$ ~ R132,
$I3$ ~ R147,
$I3$ ~ R148
```

## Full output from behaviour checker

Out from behaviour checker where substitutions are made prior to running.

```
Behaviours:
 Tau; Tau; Tau; Spn( B101 ); Tau; Spn( B118 ); Tau; Spn( B134 )

Behaviour constraints:
Paired:
Beta: B144
Behaviour: Psh ( I3 , ! int ? int end )
Paired:
Beta: B113
Behaviour: R117 ! int
Paired:
Beta: B128
Behaviour: Psh ( I3 , ! int ? int end )
Paired:
Beta: B146
Behaviour: R148 ? int
Paired:
```

```
Beta: B41
Behaviour: Tau
Paired:
Beta: B123
Behaviour: Tau; Tau; B128 ; Tau; Tau; Tau; B129 ; Tau; Tau; B130
Paired:
Beta: B57
Behaviour: Tau
Paired:
Beta: B101
Behaviour: rec B101 Tau; Tau; B108 ; Tau; Tau; B109 ; Tau; Tau;
   B110 ; Tau; Tau; B111 ; Tau; Tau; Tau; B112 ; Tau; Tau; Tau;
   B113 ; Tau; Tau; B101
Paired:
Beta: B118
Behaviour: Tau; Tau; B123 ; Tau
Paired:
Beta: B134
Behaviour: Tau; Tau; B139 ; Tau
Paired:
Beta: B139
Behaviour: Tau; Tau; B144 ; Tau; Tau; Tau; B145 ; Tau; Tau; B146
Paired:
Beta: B111
Behaviour: R115 ? int
Paired:
Beta: B110
Behaviour: Psh ( I2 , ? int ! int end )
Paired:
Beta: B129
Behaviour: R131 ! int
Paired:
Beta: B109
Behaviour: R114 ? int
Paired:
```

```
Beta: B85
Behaviour: Tau
Paired:
Beta: B108
Behaviour: Psh ( l1 , ? int ! int end )
Paired:
Beta: B112
Behaviour: R116 ! int
Paired:
Beta: B145
Behaviour: R147 ! int
Paired:
Beta: B130
Behaviour: R132 ? int
Region Constraints:
 label: l3
 regions:
  R132,R148,R131,R147,abel: l2
 regions:
  R116,R115
label: l1
 regions:
  R117,R114

Type Constraints:

check successful!
```

# Swap Delegation

## Full intermediate code

The substitutions for this code to be run through the behaviour checker are given in table 4.2.

```
tau;tau;tau;spn (B118);tau;spn (B132);tau;spn (B163)

unit <  ses R151,
unit <  ses R182,
Cswap1' ~ S120,
Cswap2' ~ S121,
Cswap3 ~ S142,
tau < B38,
tau < B74,
tau < B102,
psh ($I1$, S120) < B123,
psh ($I2$, S121) < B125,
psh ($I3$, S142) < B144,
psh ($I3$, S142) < B175,
R152 ! int < B146,
R155 ! int < B149,
R183 ! int < B177,
R186 ! int < B180,
R151 ? T160 < B145,
R154 ? T157 < B148,
R182 ? T191 < B176,
R185 ? T188 < B179,
R130 ! R131 < B127,
R153 ? $I4$ < B147,
R184 ? $I4$ < B178,
R128 ! $SWAP$ < B124,
R129 ! $LEAD$ < B126,
tau;tau;B137;tau < B132,
tau;tau;B144;tau;R150? optn {($SWAP$ ; tau;tau;tau;tau;tau;B146;
   B145), ($LEAD$ ; tau;tau;B147;tau;tau;B148;tau;tau;tau;B149;tau
   )} < B137,
tau;tau;B168;tau < B163,
tau;tau;B175;tau;R181? optn {($SWAP$ ; tau;tau;tau;tau;tau;B177;
   B176), ($LEAD$ ; tau;tau;B178;tau;tau;B179;tau;tau;tau;B180;tau
   )} < B168,
```

```
rec B118(tau;tau;B123;tau;B124;tau;tau;B125;tau;B126;tau;tau;tau;
   B127;tau;tau;B118) < B118,
$I1$ ~ R128,
$I1$ ~ R131,
$I2$ ~ R129,
$I2$ ~ R130,
$I3$ ~ R150,
$I3$ ~ R152,
$I3$ ~ R153,
$I3$ ~ R181,
$I3$ ~ R183,
$I3$ ~ R184,
$I4$ ~ R154,
$I4$ ~ R155,
$I4$ ~ R185,
$I4$ ~ R186
```

## Full output from behaviour checker

Out from behaviour checker where substitutions are made prior to running.

```
Behaviours:
Tau; Tau; Tau; Spn( B118 ) ; Tau; Spn( B132 ) ; Tau; Spn( B163 )
Behaviour constraints:

Paired:
Beta: B132
Behaviour: Tau; Tau; B137 ; Tau
Paired:
Beta: B102
Behaviour: Tau
Paired:
Beta: B144
Behaviour: Psh ( I3, +[ (SWAP;? int ? int end ) (LEAD;? ? int !
   int end end ) ][ ] )
```

```
Paired:
Beta: B137
Behaviour: Tau; Tau; B144 ; Tau; R150 ? optn [(SWAP;Tau; Tau; Tau;
    Tau; Tau; B146 ; B145) (LEAD;Tau; Tau; B147 ; Tau; Tau; B148 ;
    Tau; Tau; Tau; B149 ; Tau) ]
Paired:
Beta: B38
Behaviour: Tau
Paired:
Beta: B179
Behaviour: R185 ? int
Paired:
Beta: B126
Behaviour: R129 ! LEAD
Paired:
Beta: B125
Behaviour: Psh ( I2 , (+)[ (SWAP;? int ! int end ) (LEAD;! ? int !
    int end end ) ] )
Paired:
Beta: B146
Behaviour: R152 ! int
Paired:
Beta: B123
Behaviour: Psh ( I1 , (+)[ (SWAP;? int ! int end ) (LEAD;! ? int !
    int end end ) ] )
Paired:
Beta: B177
Behaviour: R183 ! int
Paired:
Beta: B124
Behaviour: R128 ! SWAP
Paired:
Beta: B168
Behaviour: Tau; Tau; B175 ; Tau; R181 ? optn [(SWAP;Tau; Tau; Tau;
    Tau; Tau; B177 ; B176) (LEAD;Tau; Tau; B178 ; Tau; Tau; B179 ;
```

```
    Tau; Tau; Tau; B180 ; Tau) ]
Paired:
Beta: B176
Behaviour: R182 ? int
Paired:
Beta: B127
Behaviour: R130 ! R131
Paired:
Beta: B118
Behaviour: rec B118 Tau; Tau; B123 ; Tau; B124 ; Tau; Tau; B125 ;
   Tau; B126 ; Tau; Tau; Tau; B127 ; Tau; Tau; B118
Paired:
Beta: B74
Behaviour: Tau
Paired:
Beta: B149
Behaviour: R155 ! int
Paired:
Beta: B180
Behaviour: R186 ! int
Paired:
Beta: B147
Behaviour: R153 ? I4
Paired:
Beta: B178
Behaviour: R184 ? I4
Paired:
Beta: B163
Behaviour: Tau; Tau; B168 ; Tau
Paired:
Beta: B148
Behaviour: R154 ? int
Paired:
Beta: B175
```

```
Behaviour: Psh ( l3 , +[ (SWAP;? int ? int end )  (LEAD;? ? int !
   int end end ) ][ ]  )
Paired:
Beta: B145
Behaviour: R151 ? int
Region Constraints:
 label: l4
 regions:
R154,R185,R186,R155
label: l3
 regions:
R152,R183,R184,R181,R150,R153
label: l2
 regions:
R129,R130
label: l1
 regions:
R128,R131

Type Constraints:

Paired:
Super: ses R182 sub:
 unit
Paired:
Super: ses R151 sub:
 unit
out rule
stack frame incorrect for current behaviour
ERROR: Failed Check
```

# TLS

## Full intermediate code

The substitutions for this code to be run through the behaviour checker are given in table 4.3.

```
tau;tau;tau;spn (B182);tau;spn (B210)

Ctsl1 ~ S190,
Ctsl2' ~ S216,
tau < B20,
tau < B61,
tau < B74,
tau < B111,
tau < B124,
tau < B137,
tau < B150,
tau < B177,
psh ($l1$, S190) < B192,
psh ($l2$, S216) < B218,
R201 ! int < B193,
R206 ! int < B198,
R207 ! int < B199,
R228 ! int < B220,
R229 ! int < B221,
R230 ! int < B222,
R231 ! int < B223,
R234 ! int < B226,
R202 ? T185 < B194,
R203 ? T186 < B195,
R204 ? T187 < B196,
R205 ? T188 < B197,
R208 ? T209 < B200,
R227 ? T213 < B219,
R232 ? T214 < B224,
```

```
R233 ? T215 < B225,
tau;tau;B192;tau;tau;tau;B193;tau;tau;B194;tau;tau;B195;tau;tau;
    B196;tau;tau;B197;tau;tau;tau;B198;tau;tau;tau;B199;tau;tau;
    B200 < B182,
tau;tau;B218;tau;tau;B219;tau;tau;tau;B220;tau;tau;tau;B221;tau;
    tau;tau;B222;tau;tau;tau;B223;tau;tau;B224;tau;tau;B225;tau;tau
    ;tau;B226 < B210,
$I1$ ~ R201,
$I1$ ~ R202,
$I1$ ~ R203,
$I1$ ~ R204,
$I1$ ~ R205,
$I1$ ~ R206,
$I1$ ~ R207,
$I1$ ~ R208,
$I2$ ~ R227,
$I2$ ~ R228,
$I2$ ~ R229,
$I2$ ~ R230,
$I2$ ~ R231,
$I2$ ~ R232,
$I2$ ~ R233,
$I2$ ~ R234
```

## Full output from behaviour checker

Out from behaviour checker where substitutions are made prior to running.

```
Behaviours:
Tau; Tau; Tau; Spn(B182); Tau; Spn(B210)

Behaviour constraints:
Paired:
  Beta: B194
  Behaviour: R202 ? int
Paired:
```

```
  Beta: B200
  Behaviour: R208 ? int
Paired:
  Beta: B137
  Behaviour: Tau
Paired:
  Beta: B196
  Behaviour: R204 ? int
Paired:
  Beta: B219
  Behaviour: R227 ? int
Paired:
  Beta: B193
  Behaviour: R201 ! int
Paired:
  Beta: B197
  Behaviour: R205 ? int
Paired:
  Beta: B198
  Behaviour: R206 ! int
Paired:
  Beta: B195
  Behaviour: R203 ? int
Paired:
  Beta: B124
  Behaviour: Tau
Paired:
  Beta: B177
  Behaviour: Tau
Paired:
  Beta: B199
  Behaviour: R207 ! int
Paired:
  Beta: B220
  Behaviour: R228 ! int
```

```
Paired:
  Beta: B61
  Behaviour: Tau
Paired:
  Beta: B192
  Behaviour: Psh(l1, ! int ? int ? int ? int ? int ! int ! int ?
     int end)
Paired:
  Beta: B226
  Behaviour: R234 ! int
Paired:
  Beta: B74
  Behaviour: Tau
Paired:
  Beta: B111
  Behaviour: Tau
Paired:
  Beta: B225
  Behaviour: R233 ? int
Paired:
  Beta: B20
  Behaviour: Tau
Paired:
  Beta: B223
  Behaviour: R231 ! int
Paired:
  Beta: B150
  Behaviour: Tau
Paired:
  Beta: B224
  Behaviour: R232 ? int
Paired:
  Beta: B182
  Behaviour: Tau; Tau; B192; Tau; Tau; Tau; B193; Tau; Tau; B194;
     Tau; Tau; B195; Tau; Tau; B196; Tau; Tau; B197; Tau; Tau; Tau
```

```
      ; B198; Tau; Tau; Tau; B199; Tau; Tau; B200
Paired:
  Beta: B218
  Behaviour: Psh(l2, ? int ! int ! int ! int ! int ? int ? int !
      int end)
Paired:
  Beta: B221
  Behaviour: R229 ! int
Paired:
  Beta: B222
  Behaviour: R230 ! int
Paired:
  Beta: B210
  Behaviour: Tau; Tau; B218; Tau; Tau; B219; Tau; Tau; Tau; B220;
     Tau; Tau; Tau; B221; Tau; Tau; Tau; B222; Tau; Tau; Tau; B223
      ; Tau; Tau; B224; Tau; Tau; B225; Tau; Tau; Tau; B226
Region Constraints:
 label: l2
 regions:
R228,R230,R231,R229,R234,R232,R227,R233
label: l1
 regions:
R205,R201,R204,R207,R202,R203,R206,R208

Type Constraints:


check successful!
```

# Bibliography


[1] C. Spaccasassi and V. Koutavas, "Type based analysis for session inference."

[2]

[3] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Denilou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro, "Foundations of behavioural types." `http://www.behavioural-types.eu/publications`, 2014.

[4] T. Amtoft, F. Nielson, and H. R. Nielson, *Type and Effect System - Behaviours for Concurrency.* Imperial College Press, 1999.

[5] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani, "Foundations of session types," 2009.

[6] OCaml. `https://github.com/ocaml/camlp4/wiki`.

[7] F. Pottier and Y. Régis-Gianas, "Menhir reference manual." `http://pauillac.inria.fr/~fpottier/menhir/manual.pdf`.

[8] T. Team. `http://toss.sourceforge.net/ocaml.html`, 2013.

[9] Yan. `http://yansnotes.blogspot.ie/2014/11/menhir.html`, 2014.

[10] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml.* O'Reilly Media, 2013.

[11] *Functional programming using Caml Light*, ch. 6.

[12] B. C. Pierce, *Types and Programming Languages.* The MIT Press, 2002.

[13] "Core_kernel library." https://ocaml.janestreet.com/ocaml-core/111.28.00/doc/core_kernel/.