Formal Verification of Session Types

by

Théa Johnson,

Thesis

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Bachelor's of Arts

University of Dublin, Trinity College

April 2016

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Théa Johnson

April 11, 2016

Acknowledgments

...ACKNOWLEDGMENTS...

Théa Johnson

University of Dublin, Trinity College
April 2016

Formal Verification of Session Types

Théa Johnson, B.A.

University of Dublin, Trinity College, 2016

Supervisor: ...

...ABSTRACT...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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], and to implement a behaviour checker based on the designs described in this paper.

1.1 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 we can show 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 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 type systems of all modern languages.

1.2 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.

1.3 Background

Traditional type systems embedded into programming languages focus on the computations and what they should produces. Session type systems aim to embed session types that describe both the sequence and they type of messages that are been transmitted on communication channels into modern programming languages. This allows for the verification of the communications on a channel since the session type describes the protocol of the channel.

1.3.1 Main Session Type Approaches

According to Hüttle et al. [2] the main approaches to session types are as follows.

Session types are usually associated with binary communication channels where the two ends using the channel see it as complementary types. It is then possible to use static type checking to ensure that the communications on the channel are in accordance with the protocol. *Multiparty session types* extend binary session types to allow for more than two programs to communicate. *Conversation session types* unify local and global multipary types and allow for unspecified numbers for communication. *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 introduces a new idea for a design approach to Binary Session Types which uses effects. A high level language is developed where communication protocols can be programmed and checked statically. In this paper [1] the approach is applied to ML_s, a core of the language ML with session communication.

The approach suggested separates traditional 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 [3]. Types allow for clear representation of program properties and enable modular reasoning. The effects are essentially behaviour types that record the actions of interest that will happen at run time. From this system a complete behaviour inference algorithm is obtained, this extracts a behaviour for a program providing it respects ML types. In this level the programs are typed against both an ML type and a behaviour. Session protocols are not considered here and so endpoints are given a type ses^{ρ} instead (see 2.3).

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 2.3).

This level is inspired by the work done by Castagna et al. [4]. 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 by this system 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 stacked interleaving of sessions 2.3.

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

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.

2.2 The Fist Level

At this level the type and effect system of Amtoft, Neilsen and Neilsen 1999 [3] is extended to session communications in ML_s. The type and effect system consists of constructions of judgments of the form $C; \Gamma \vdash e : T \triangleright b$. In this statement C represents the constraint environment which is used to relate type level variables to terms and so enables session inference. Γ represents the type environment which is used to bind program variables to type schemas. To read this judgment we would say that expression e has type T and behaviour b under type environment Γ and constraint environment C.

In the system designed in the paper an ML_s expression can have either a standard ML type or a session type. Session types are of the form ses^ρ where ρ is a static approximation of the location of the endpoint. Functional types have an associated behaviour β and type variables α are used for ML polymorphism.

Figure 2.1: Syntax of types, behaviours, constraints and session types

Figure 2.2: Type and Effect system for ML_s Expressions omitting rule for pairs

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

The rules for the type and effect system proposed are given in fig. 2.2. These are made of of the judgments described above and requirements for the rule. These rules say that if we have a judgments of the form given above the line and if the requirements beside the rule (if they exist) are met then the judgment below the line will be valid.

2.2.1 explain the rules

2.2.2 Type Schemas, Locations and Region Variables

Through the constraint environment (C) region variables (ρ) are linked to region constraints. These region constraints are produced during pre-processing. They are annotations to the program that identify the textual source of endpoints.

For example if $req - c^l$ is called we know from l where in the code it was 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.

If we have an expression with type Ses^{ρ} which evaluates to p^l then a region constraint must exist to link l to ρ. This takes the form of $C \vdash \rho \sim l$ which says that ρ and l are linked under constraint environment C. This tells us that p was generated from the location int the code referenced by l.

If we were to look up this location we would find one of $req - c^l, acc - c^l orresume - c^l$ where c references the channel on which the communication will take place. These primitive functions are typed by the rule TConst given in fig. 2.2

They type schemas of these primitives are given in fig. 2.3. In the case of $req - c^l$ a new session is started on the static endpoint l. In order for it to be type-able C must contain its effect which is $push(l : \psi) \subseteq \beta$. In the stack frame ψ is the session

$$req - c^l : \forall(\beta\rho\psi : push(l : \psi) \subseteq \beta, \rho \sim l, c \sim \psi).Unit \xrightarrow{\beta} Ses^{\rho}$$

$$acc - c^l : \forall(\beta\rho\psi : push(l : \psi) \subseteq \beta, \rho \sim l, \bar{c} \sim \psi).Unit \xrightarrow{\beta} Ses^{\rho}$$

$$ressume - c^l : \forall(\beta\rho\rho' : \rho?\rho' \subseteq \beta, \rho' \sim l).Ses^{\rho} \xrightarrow{\beta} Ses^{\rho'}$$

$$recv : \forall(\alpha\beta\rho : \rho?\alpha \subseteq \beta cfd(\alpha)).Ses^{\rho} \xrightarrow{\beta} \alpha$$

$$send : \forall(\alpha\beta\rho : \rho!\alpha \subseteq \beta cfd(\alpha)).Ses^{\rho} \times \alpha \xrightarrow{\beta} Unit$$

$$deleg : \forall(\alpha\rho\rho' : \rho!\rho' \subseteq \beta).Ses^{\rho} \times Ses^{\rho'} \xrightarrow{\beta} Unit$$

$$sel - L : \forall(\alpha\rho : \rho?L \subseteq \beta).Ses^{\rho} \xrightarrow{\beta} Unit$$

Figure 2.3: Type Schemas

Figure 2.4: Abstract Interpretation Semantics

variable that represents the session type of endpoint l. C must also record that the region variable ρ is linked to l and that the "request" endpoint of c, the channel, has session type ψ.

The remaining type schemas are read in a similar way.

2.3 The Second Level

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

$$\eta ::= end | !T.\eta | ?T.\eta | !\eta.\eta | ?\eta.\eta | || \psi$$

Either communication on the endpoint is finished (end) or more communications are going to take place. These include sending or receiving a confined type. Confined here meaning that this type will not effect the frames on the stack and any new endpoints created will be consumed. Endpoints can also be sent over other endpoints for delegation ($!\eta.\eta$) and can be resumed ($?\eta.\eta$). Non deterministic selection of a label L_i indicated that the session type associated with it, η_i, is to be followed next. The offer of external choice to the communication partner is also supported.

2.3.1 explain rules

Chapter 3

My Contribution

My contributions to this project are based on the second level. These include 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 will convert the input code, which will consist of one behaviour and one constraint, into the types defined in OCaml to represent the behaviours and constraints. A tuple of these will then be returned and passed into the behaviour checker where the rules from 2.4 will be applied to verify that the program will in fact communicate as expected.

One limitation of the checker that is worth noting is that it will not infer types and so these must be explicitly stated 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. To develop an easy to parse version of this syntax Greek letter have been removed, keywords or individual letters have been chosen to replace them. The substitutions made can be seen in the given syntax for the input code in section 3.2.4 and the table below detailing the syntax of the substitutions where their syntax is not directly stated in the grammars given.

Type Variables	α	[T][A-Z a-z 1-9]+
Behaviour Variables	β	[B][A-Z a-z 1-9]+
Session Variables	ψ	[S][A-Z a-z 1-9]+
Region Variables	ρ	[R][A-Z a-z 1-9]+
Labels	l	$[A-Z a-z 1-9]+$

3.1.2 Challenges

The design of the text based syntax was relatively simple and posed few challenges.

3.2 Lexer and parser

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. 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 $ for labels, calls a second lexing function to deal with labels.

3.2.2 Parser

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

In the parser first the tokens are declared for every relevant grouping of input. Then the structure of the accepted input and the code to be invoked when that input is encountered are specified. The structure of the input can be seen in 3.2.4.

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 from scratch these took some time to get used to.

Another challenge encountered with the lexer was parsing the end of file. An unknown issues 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.

Finally initially there were some errors with shift/reduce conflicts in the parser. 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. This was that it was unclear in the case $rec < behaviourVariable >< behaviour >; < behaviour >$ if the recursive behaviour was in the sequence or if the sequence was the behaviour associated with the recursion.

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 all cases in this description of the grammar the 'Strings' are the input as it appears in the file and anything inside <> is either another type from the grammar or a variable the syntax of which is given in 3.1.

⟨*parse_behaviour*⟩ ::= ⟨*behaviour*⟩ ⟨*constraint*⟩ EOF

Behaviour

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

⟨*behaviour*⟩ ::= ⟨*behaviourVariable*⟩
| 'tau'
| ⟨*behaviour*⟩ ';' ⟨*behaviour*⟩
| 'chc' (⟨*behaviour*⟩,⟨*behaviour*⟩)
| 'rec' ⟨*behaviourVariable*⟩ (⟨*behaviour*⟩)
| 'spn' (⟨*behaviour*⟩)

| ‘`psh`’ (⟨*lable*⟩,⟨*sessionType*⟩)
| ⟨*region*⟩ ‘`!`’ ⟨*behaviourVariable*⟩
| ⟨*region*⟩ ‘`?`’ ⟨*behaviourVariable*⟩
| ⟨*region*⟩ ‘`!`’ ⟨*region*⟩
| ⟨*region*⟩ ‘`?`’ ⟨*lable*⟩
| ⟨*region*⟩ ‘`!`’ ⟨*lable*⟩
| ⟨*region*⟩ ‘`?`’ option[⟨*oplist*⟩]

⟨*oplist*⟩ ::= ‘`(`’ ⟨*lable*⟩ ‘`;`’ ⟨*behavioiur*⟩ ‘`)`’ ⟨*opt_feild*⟩

⟨*opt_feild*⟩ ::= ‘`,`’ ‘`(`’ ⟨*lable*⟩ ‘`;`’ ⟨*behavioiur*⟩ ‘`)`’
| ϵ

Constraints

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

⟨*constr*⟩ ::= ⟨*bType*⟩ ‘<’ ⟨*bType*⟩
| ⟨*behaviour*⟩ ‘<’ ⟨*behaviour*⟩
| ⟨*region*⟩ ‘`~`’ ⟨*regionVar*⟩
| ⟨*channel*⟩ ‘`~`’ ⟨*sessionType*⟩
| ⟨*channelEnd*⟩ ‘`~`’ ⟨*sessionType*⟩
| ⟨*contr*⟩ ‘`,`’ ⟨*constr*⟩
| ϵ

Types

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

⟨*bType*⟩ ::= ‘`unit`’
| ‘`bool `’
| ‘`int`’
| ‘`pair`’ ‘`(`’ ⟨*bType*⟩ ‘`;`’ ⟨*bType*⟩ ‘`)`’
| ‘`funct`’ ⟨*bType*⟩ ‘`->`’ ⟨*bType*⟩ ‘`-`’ ⟨*behaviourVariable*⟩

| ‘`ses`’ ⟨*region*⟩
| ⟨*TVar*⟩

Region Variables

Regions are very simple and either consist of a region variable or a label (see 3.1 for syntax).

⟨*regionVar*⟩ ::= ⟨*lable*⟩
| ⟨*region*⟩

Session Types

⟨*sessionType*⟩ ::= ‘`end`’
| ‘`!`’ ⟨*bType*⟩ ⟨*sessionType*⟩
| ‘`?`’ ⟨*bType*⟩ ⟨*sessionType*⟩
| ‘`!`’ ⟨*sessionType*⟩ ⟨*sessionType*⟩
| ‘`?`’ ⟨*sessionType*⟩ ⟨*sessionType*⟩
| ‘`(+)`’ ‘`[`’⟨*sesOpL*⟩‘`]`’ ‘`(`’⟨*lable*⟩ ‘`;`’ ⟨*sessionType*⟩‘`)`’
| ‘`+`’ ‘`[`’⟨*sesOpL*⟩‘`]`’ ‘`[`’⟨*sesOpL*⟩‘`]`’
| ⟨*SVar*⟩

⟨*sesOpL*⟩ ::= ‘`(`’⟨*lable*⟩ ‘`;`’ ⟨*sessionType*⟩‘`)`’ ⟨*ses_opt_field*⟩

⟨*ses_opt_field*⟩ := ‘`,`’ ‘`(`’⟨*lable*⟩ ‘`;`’ ⟨*sessionType*⟩‘`)`’
| ϵ

3.3 OCaml Types

In order for the parser to have the correct types for dealing with behaviours and constraints 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.

These types take 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};;
```

you can clearly see 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.

3.4 Behaviour Checker

The behaviour checker is implementing 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. We then had to look at the behaviour read in from the file and check it in relation to these constraints.

3.4.1 Storing the Constraints

The constraints that are 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 not stored.

Behaviour Constraints

These constraints are of the form

$$b \subseteq \beta$$

where b is a behaviour and β is a behaviour variable. These constraints are used in the Beta and Rec rules. In the Beta rule ($\triangle \models \beta \rightarrow c\triangle \models b$ if $C \vdash b \subseteq \beta$) the constraints are used to replace the behaviour variable beta with each behaviour associated with it and to check that each of these are valid.

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. In this case the left hand side of that rule is replaced with Tau.

The decision was made to store the behaviour constraints as a hash table. The key is the right hand side of the constraint β and the value is the left hand side (the behaviour associated with beta). In this was 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 β 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 recursion behaviour.

Type Constraints

Type constraints are of the form

$$T_1 <: T_2$$

and are used in the Out Rule. They are used according to semantics that state that if the constraint 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 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 function.

If both types are of the form Ses^ρ then if there exists a region constraint linking the ρ'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 we first search for the right hand side of the constraint we wish to check. Then we get the list of associated subtypes for that type and recursively search for these as keys until either we run out of types to search for or we find the type we are looking for.

Since type constraints are also reflexive we always check if the two types are equal first.

Region Constraints

Region constraints take the form of either

$$\rho \sim \rho'$$

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 simply 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 $\rho_1 \sim \rho_2$, $\rho_2 \sim \rho_3$ and $\rho_3 \sim l1$ tells us that ρ_1, ρ_2 and ρ_3 are all linked to $l1$.

Due to this region constraints are stored as a list of tuples which each consist of a label and a hash table with region variables as keys and unit as 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.

Storing the constraints in this way in a single pass through the constraint list was challenging. It is achieved by first creating a new tuple consisting of either a label or a None and a hash table with either one or two values. 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. The are merge together with the new element to form a single list element and added back to the new list which is returned.

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 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 is used to ensure that each label is only ever pushed to the stack once. The continuation is used to keep track of any behaviours that need to be dealt with once the current behaviour is finished with.

The initial function called is a wrapper function that calls to a check step function with the first tuple from the list. The function is set up this way so that if the need arose to read in more than one behaviour from a file it would be supported.

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 applies the end rule) if it does it removes this frame and then it continues to check the to see which of the rest of the rules apply.

The behaviour of the Beta rule has already been briefly described. A diagram detailing the behaviour of the Sequence can be seen in fig. 3.1. As you can see from

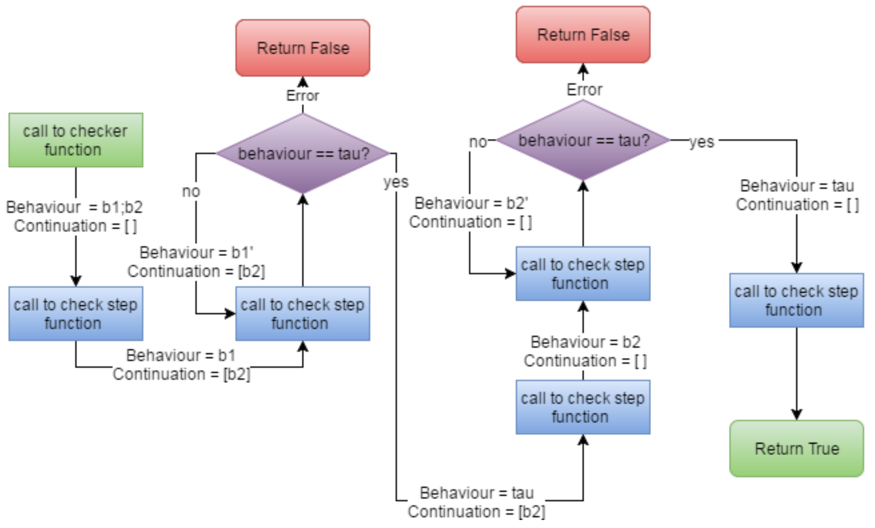

Figure 3.1: Application of Sequence Rule

this the continuation is used to store the second behaviour in the sequence while the first one is been dealt with.

The other rules are applied in a similar way.

3.4.3 Results

The behaviour checker will output for us the results of the check. In the case of the check failing it will also inform us of the rule on which the check failed but not the location in the source code.

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

3.4.4 Challenges

The main challenges 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.

Other challenges 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 included functions that made for a much nicer implementation but the hash tables were missing the functionality to return lists of all bindings to a key. 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 was also took quiet some time since it is rather complicated code and pushed my knowledge of the OCaml language to it boundaries.

Chapter 4

From Start to Finish

Appendix

...

Bibliography

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

[2] 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.

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

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