Efficiently Checking Subtyping of Session Types

Sunday 16$^{\text{th}}$ January, 2022 - 09:17

Paul Houssel
University of Luxembourg
Email:paul.houssel.001@student.uni.lu

This report has been produced under the supervision of:
Ross James Horne
University of Luxembourg
Email: ross.horne@uni.lu

Abstract—**This document is the final report of the Bachelor Semester Project of the student Paul Houssel which was conducted with the help of his tutor Ross James Horne. This project is part of the Bachelor in Computer Science program at the University of Luxembourg during the fifth semester of study. This project seeks to define a solution to check subtyping of session types as well as to extend a subtyping checking tool developed by Lorenzo Bacchiani, a researcher at the University of Bologna, Italy.**

1. Introduction

In system communications, two or more systems are communicating to each other in a session in order to transmit information. For avoiding common mistakes in communications e.g. in secure communications and distributed systems, it is of highest importance to guarantee data integrity and communication safety. Session types are defining patterns of communications as well as behaviours these interactions shall respect. What message types shall be received?; What types shall be sent?; how shall the system behave after receiving a certain message type ? All these system constraints are specified by session types, forming a communication protocol. Session types allow us to model protocols, they can be seen as protocol types. It is a type-theoretic specification of protocols. Session types originate in pi-calculus, developed by Kohei Honda in 1993 [5]. This project will treat about it's subtyping, which allows us to define set relations between different session types. While two session type can be compatible, a type can be a subtype of a supertype, which implies that operating on elements of the supertype can also be done on elements of the subtype. Using this concept, we are able to use simpler to implement subtypes, while keeping the safety attributes guaranteed by the supertype. The subtyping tool [1] checks has three different algorithms that are able to check if a subtyping relation holds, accordiing to different subtyping definitions, between two session types. By simulating the communications, the used algorithms verifying if this subtyping relation holds. Finally, the main goal of this project is to extend this software by implementing and adding to the tool a proof concept for proving subtyping. The design and presentation of this proof system consist of the scientific deliverable.

2. Project description

2.1. Domains

2.1.1. Scientific.

Session Types. Session Types are, types formalising interactions between participants of a communication system, they are described using a syntax which will be defined in this report.

Subptying. In Computer Science, and moreover in programing, subtyping, is defined as the relation between two different datatypes which are linked together (super-type and sub-type). A program working with elements built of the super-type, can also work with elements of the sub-type. In general, the supertype-subtype relation, is based on the Liskov's substitution principle, Let $\phi(x)\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)\phi(y)$should be true for objects y of type S where S is a sub-type of T.

Network Protocol. Set of different rules for a specific type of communication.

2.1.2. Technical.

Haskell. Haskell is a high level programing language, which has the particularity to be a functional programing language. This language is used to implement the session subtyping algorithm, it was chosen because this language was already used by Lorenzo Bacchiani to implement ohter subtyping checking algorithms. Furthermore, it allows us to implement the algorithm by remaining closely to the formal definitions.Functional programming is based on the relation between types and logic, a key asset needed to conduct a prove on the session types.

Parser. A parser allows us to process a string by a set of syntax rules formed by a grammar, in to a structured set of elements (formalizing text). A parser is used to formalise the input given by the user to the algorithm. A natural language text is formalised such that a machine can work with it.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. Session types are used to model and structure communications between different systems. They are used to avoid common mistakes when designing and implementing protocols, e.g. in secure communications and distributed systems, it is also crucial in order to guarantee data integrity and communication safety. While designing a tool for checking protocols, we need to ask ourselves. How can we design a focused proof system for checking subtyping relations between session types ? That is a system for efficiently proving that a type is a subtype of it's supertype.

2.2.2. Technical deliverables. The technical deliverable is based on the session subtyping tool built by Lorenzo Bacchiani [1] in the python programming language using executable Haskell programs. This sofware is licenced under the *MIT Licence*, allowing the tool to be copied, modified and published [7], under the condition to include the Licence in this extension source code. His tool allows to apply different subtyping algorithms that will simulate subtyping in order to check subtyping of two session types. We will design, develop and integrate into this tool a new asynchronous session subtyping algorithm proven by my tutor Ross James Horne. By extending it, the tool will support parallel session types, acting as internal communications on one side of the protocol. The proof system designed and presented in the scientific part of this project will be used in this tool. With the help of the new parser generator, the session types will be handled by the algorithm, in order to verify if a subtyping relation holds. Furthermore, it will also be possible to visualise internal communications of these session types under the form of a graph. Once integrated into the existing software, the already existing GUI, acts as an input interface of the algorithm. As an output, a readable text file including the algorithms taken steps and result is generated. The source code of the extended tool is freely available on github [8] and licenced under the *MIT Licence*.

3. Pre-requisites

3.1. Scientific pre-requisites

To successfully understand the study of subtyping proof concepts used in session subtyping, a background in linear logic and discrete mathematics is needed in order to understand the functioning of the subtyping algorithm. It is also helpful to have some basic networking concepts.

3.2. Technical pre-requisites

As the technical part mainly involves coding, the technical pre-requisites are functional programming in the Haskell programming language.

4. Parallel Session Types and Subtyping

4.1. Requirements

The primary purpose of this deliverable is the study and explanation needed to answer the following scientific question: How can we design a focused proof system for checking subtyping relations between session types ? The production part of this deliverable has the following requirements,

- [4.3.1] The deliverable shall define Session Types
- [4.3.2] The deliverable shall define Session Subtyping
- [4.3.3] The deliverable shall define Parallel Session Types
- [4.3.4] The deliverable shall define Parallel Session Subtyping and a proof system to check it

4.2. Design

The scientific report should briefly give an introduction to the concept of session types and the problem and importance of sub-typing in order to understand the following proof system for session types. All concept are supported by concrete and visual examples created with the supbtyping tool [1]. Finally, while we present the proof concept to check for sub-typing, a parallel with the technical deliverable is made. Since the presented algorithm will be used in the technical deliverable, it will be important for understanding that part. Before passing on onto the actual proof system for checking subtyping focused in this report, the functioning of the existing subtyping checking algorithms are explained. This will contribute to contrast their differences and how they extend each other, to understand the main contribution of this new algorithm.

4.3. Production

4.3.1. Session Types. It is firstly important to define the term of sessions, as stated in the publication "Sessions and Session Types: An Overview"[2], it is a “unit of information exchange between two or more communicating systems”. Sessions are defining a series of network interactions in the common goal of interchanging information. In system communications, two or more systems are communicating to each other in a session in order to transmit information. For avoiding common mistakes in communications e.g. in secure communications and distributed systems, it is of highest importance to guarantee data integrity and communication safety. [4]

Employed in Sessions, Session types are defining patterns of communications as well as, behaviours the interactions shall respect. What message types shall be received; what types shall be sent; how shall the system behave after receiving a certain message type ? All these possible system interactions are specified into constraints by Session types, forming a communication protocol. [1] It is a type-theoretic specification of protocols. Session types originate in pi-calculus, a theoretical programming language, used to describe interacting and parallel computing agents.

In this report we will treat two different kinds of communications, Asynchronous and Synchronous communications. In an asynchronous communication, not every participant of the communication must be present or available at the same time for the communication to take place. A message waiting for an answer will not block the rest of the protocol to function correctly, it is said that the communication does not happen in real time. In synchronous communication however, the communication happens in real time, a message waiting for his response will block the rest of the communication.

Session types are defined in terms of a language having it's proper syntax. The language forming correct session types can be generated by a regular grammar G, defining it's syntax to build the terms of session types,

$$G \to \mu D.B| + \{A; G, A; G\}|\&\{A.G; , A; G\}|$$

$$!A; G|?A; G|B|end$$

$$A \to a..z$$

$$B \to A..Z$$

The "!" and "?" symbol respectively indicate weather to send or receive, it is followed by a a identifier (lowercase) defining what shall be send or received. '$\mu B.$' is denoting a recursion defined a uppercase letter, this letter is another session type. The operators + and *'&'* are respectively standing for making a choice to send either one type or another type and having a choice between receiving a first type or a second type. '*End*' defines the end of the session.

Let's look at an example in order to illustrate a possible session type. Let *SERVER* be the university's server, and *USER* a user trying to submit his project to the server [3]. The interactions between the client and the server will be described by the following session type in the view of the User,

$$!String.\&\{?file; end, ?string; end\}$$

Using "!String" designates that the USER shall send a string message,"?file" states that the user will receive a file, "&" indicates that the USER can either receive a file or the session is ended. In this situation the user sends his credentials in order to authenticate himself, the server will either accept them and continue the session or end it in the case of an authentication failure. If accepted, the user will send a file to the server. In the View of the SERVER we have the following session type,

$$?String. + \{!file; end, !string; end\}$$

We can notice that the "!" symbols are interchanged with "?" and the "&" are interchanged with "+". This is in fact a basic feature of session types called duality of sessions, it is allowing communication safety [2], in binary communications it is even allowing deadlock-freedom. A message sent, shall be received, and vice-versa.

By converting a regular grammar G into a Finite State Automata we can visualise session types. Let's come back to our example, using the session type of the USER, considered as a string $\in L$ (language defined by Grammar G). The corresponding finite state automata is depicted in figure 1, it is generated by the subtyping checker tool [1].

4.3.2. Session Subtyping. A session subtype is a session type related to a corresponding session Supertype by substitutability. The concept of substitutability implies that if a protocol is compatible and working with a supertype, then it is also for it'subtype. The subtype can be used instead of the supertype in any context. We will use the following notation, $U \leq U'$, U is a subtype of the Supertype U'.

In a binary communication, the two end points shall have dual session types for assuring Deadlock-freedom, it is a hard to respect prerequisite. Let's say we want to implement a process P of session type U, we can implement process P' of session type U', if and only if U' is a subtype of the supertype U. This can allow to escape the rigid constraints. Proceeding this way, we need to be able to check subtyping. This can be done using subtyping algorithms. Algorithms differ in case of synchronous and asynchronous communication protocols. Using subtyping in the field of session types, session types become more powerful.

Let's come back to our example of a student communicating with the university's website, wanting to search a specific project on the website. The server is now able to deal with two different requests, one for submitting the name of the project and one for deleting a project of the list. A negative response is sent when the file name is wrong, or the file wished to be deleted does not exist. We will use the tool developed by Lorenzo Bacchiani [1] to visualise the session type in the form of a state automata.

User Session type (Figure 2):

$$recX.+\{submit; \&\{ok; X, ko; X\}, delete; \&\{ok; X, ko; X\}\}$$

After receiving the request, the server either responds with okay or not okay. Once it responded, it is able to receive another request. By finding the dual of the USER type, we can obtain the SERVER type,

Server Session type (Figure 3):

$$recX.\&\{submit; +\{ok; X, ko; X\}, delete; +\{ok; X, ko; X\}\}$$

We will now create another session type of the USER, USER' , the user is now only able to delete a file, the server responds either okay, not okay or don't know in case he is not sure the deletion was complete. We obtain, the result in figure 4.

For a client type to be compliant to a server type these conditions need to be fulfilled :

1) Every message sent by the client can be received by the server
2) The server nor the client are blocked while receiving a message

We can observe that USER' session type is compliant to the SERVER session type, thus USER' type is a synchronous subtype of USER type.

In asynchronous communications, compliance between two end-points is achieved if all messages eventually are received. The tool used to create the finite state Automata is also able to check subtyping using subtyping algorithms. In order to check subtyping, reflexivity and transitivity of session types needs to hold. An algorithm receives two types, a session type and it's supposed supertype, as an output it will tell us if one type is the subtype of the other. The tool is composed of three different simulation algorithms that consider the subtype and the supertype as state automatas, starting from the start state, the simulation matches every transition from the subtype automata to the supertype automata. If even one transitions does not match, the simulation fails, the pretended subtype is not a subtype of the pretended supertype. The algorithm will not go in an infinite path, as it is detected. Let's run the synchronous subtyping simulation on USER' and USER session types, we obtain the result in figure 5. The simulation was a success, USER' is indeed a subtype of USER.

Let R be the reflexive relation between the two session types. We can formalize the simulation as follows in pseudocode:

1) Gay Hole Version

$$\text{if}(s,t) \in R$$

$$\text{if} s \xrightarrow{?\text{a}} s'$$

$$\exists t' \text{such that } t \xrightarrow{?a} t' \text{ and } (s',t') \in R$$

$$\text{R is a simulation and } U \leq V \text{ if and only if}$$

$$\exists \text{R simulation such that} (u,v) \in R$$

2) Kozen Version, alternative extension

$$\text{if}(s,t) \in R \text{ and if} t \xrightarrow{!b} t'$$

$$\text{then } \exists s' \text{ such that } s \xrightarrow{!b} s' \text{ and } (s',t') \in R$$

For each different definition, there exists a corresponding algorithm. The *Kozen* version is characterized by verifying the simulation in both directions (from the subtype to the supertype and vice-versa).

In contrast to simulation tree-based algorithms, the new algorithm presented in this report will moreover consist of searching for the correct proof, a proof search algorithm.

4.3.3. Parallel Session Types. As this new algorithm is in a certain way more powerful, it applies to a wider range of session types.

Let's first introduce the concept of parallelism in terms of session types. A participating agent in a communication protocol, may not only communicate with the other participants, but also with other internally linked components or sub-agents. For instance, a user is communicating with a web server, the webserver is giving the user the asked data which is stored on the web server. We now have two session types in the communication protocol (USER and SERVER), the communication is assured to be deadlock free. However what happens when the data is stored on an external database, the server is relying on a parallel thread in order to respond to the user's demand. We can safely replace the SERVER with two different types (SERVER' and SERVER"), if and only if

$$SERVER'|SERVER" \leq SERVER$$

As subtyping holds, the deadlock freedom that is guaranteed under $SERVER$ will also be granted with the user of $SERVER'|SERVER"$. The single old components is safely replaced by the two new components. To define two parallel session types we use the following symbol "|". The dual of "|" is defined by the symbol "$" [6].

In another case, two agents communicate on a network,

$$\text{AGENT 01:} ?a; end|!a; end$$

$$\text{AGENT 02:} !a; end$$

The first agent has two internal threads, it can either receive "a" from it's internal component or from the second agent.

4.3.4. Parallel Session Subtyping. A subtyping relation of parallel session types can be formalized as follows [6],

$$\text{if } (s,t) \in R \text{ then } \exists \text{u s.t } t \rightarrow u$$

1) Verifying the relation from the subtype to the supertype

$$\text{if } s \xrightarrow{?a} s' \text{ then } \exists u' \text{ s.t } u \xrightarrow{?a} u' \text{ and } (s',u') \in R$$

2) Verifying the relation from the supertype to the subtype

$$\text{if } u \xrightarrow{!a} u' \text{ then } \exists s' \text{ s.t } s \xrightarrow{!a} s' \text{ and } (s',u') \in R$$

and there exists t'such that $u' \rightarrow t'$ and $(s',t') \in R$.

As in the simulation based algorithms we saw earlier, there exists two variation, the first one is checking the relation from the subtype to the supertype, the second one the other way around.

In order to check for subtyping when parallel session types are used, a proof system developed by Ross Horne in his paper *Session Subtyping and Multiparty Compatibility using Circular Sequents* [6] can be applied. By considering this proof system, a proof search algorithm can be implemented. To prove that a subtyping relation holds, a set of rules can be applied onto the session types. The set of rule used for proving subtyping are defined in figure 6. These rules lead to a set of premises on which conclusions can be drawn. This guide leading to a proof can be visualised using an inference rule tree.

Let's say we want to prove D by knowing A, we apply a rule β onto A, by definition of this rule, we obtain,

Tree 01:
$$\frac{B \quad C}{A} \text{ [RULE } \beta]$$

Tree 02:
$$\frac{D}{A} \text{ [RULE } \beta]$$

Applying the rule β implies the premise $((B \wedge C) \vee D)$ drawn from A. It is therefore proved that $A \quad \Gamma \Rightarrow \quad D$, as at least one obtained tree proves it. Every tree can be considered as a formal proof for subtyping. Obtaining multiple tree's corresponds to finding multiple logical premises that are implied by the initial input.

The algorithm presented in this report, will apply these rules to finally obtain a set of proofs. It consists of every possible set of logical premises that can be drawn from the input. A session type is considered as a sequent, applying a rule on it, implies a set of new sequents on which can be applied the same rules again. According to the Curry-Howard correspondence [9], rule can be considered as function proving that A implies B, constructed as as proof of B from A. Starting from the initial sequent, using these rules, it is proven that this sequents corresponds to every inferring sequents (all the sequents obtained at the end of the algorithm). We cannot apply rules between two sequents. Each sequent in the set of sequents is independent. A sequent is a set of logical assertions, in our case it is a single or multiple session types. E.g

$$A_1, A_2 \vdash B \mathrel{\widehat{=}} \text{If } (A_1 \wedge A_2) \text{ then } B \text{ is } TRUE$$

Finally, all the possible premises are obtained, it is verified if these premises imply a subtyping relation. Some of them might, some of them might not, it is verified using the *OK* rule in figure 6. This rule holds whenever the sequent is only composed of *end*'s. Once no more rule can be applied, the algorithm has generated a list of tree's each having a list of branches, only the upper-most branches (sequents) are taken into account.

Final result:

$$\overbrace{(Sequent_1 \wedge \quad \underbrace{Sequent_2}_{Sessiontype_1...} \quad)}^{\text{Tree 01}} \vee \overbrace{(Sequent_1)}^{Tree02} \vee \overbrace{(Sequent_1)}^{\text{Tree 03}}$$

Every branch, applying the *OK* rule either implies $TRUE$ or $FALSE$ in regards to if the subtyping relation holds. If the final result evaluates to $TRUE$ then the subtyping relation does hold, otherwise it does not hold.

Depending which definition of subtyping we want to prove (either 1. or 2., as defined in the first paragraph of section *4.3.1*), the supertype is dualized for definition 1., for definition 2., the subtype is dualized. The algorithm will apply the first rule on a sequent containing both the subtype and the dualized supertype, or vice-versa, the algorithm applies the rule in specific way,

1) check whether the JOIN and TIMES rules can be applied and apply them everywhere. This will generate a set of smaller sequents, where there are no "internal choices" or the parallel composition "times" (x) at the top level.
2) Try to apply the MEET, PREFIX and PAR rules. To do so we select any type in the sequent which is either of the form ?a;T, an external choice between inputs, or a par (upside down in the paper). We then search for a way to apply the MEET and PAR rules such that a PREFIX rule may eventually be applied. There exists multiple ways to proceed, applying the rules, the order does matter a lot. Every possible order is therefore represented in a different tree.
3) Once no more rule can be applied, every sequent is evaluated using the *OK* rule. As explained in the previous figure, if there exists at least one tree which has every sequent composed of only *end's* then the subtyping relation holds.

Examples and explanation of the rules used by the algorithm,

1) Times: $\forall$ parallel session types in a sequent, $(T|U)$ implies (T, U). One sequent corresponds to a new sequent. E.g,

$$\frac{?a; end, !b; end, ?a; end}{?a; end, !b; end|?a; end}\ TIMES$$

2) Join: $\forall$ session types in a sequent, $A; +\{T, U\}$ implies $A; T \wedge A; U$. E.g,

$$\frac{?a; end, !b; X \quad ?a; end, !c; X}{?a; end, +\{!b; X, !c; X\}}\ JOIN$$

3) Meet: $\forall$ session types in a sequent, $A; \&\{T, U\}$ implies $A; T \quad \vee \quad A; U$. Two new tree's are generated, since one of these premise needs to hold $(1 \vee 2)$. E.g,

$$\frac{A; ?a; end}{A; \&\{?a; end, ?b; X\}}\ MEET$$

Or,

$$\frac{A; ?b; X}{A; \&\{?a; end, ?b; X\}}\ MEET$$

4) Prefix: For every first action of a session type in a sequent, if $\exists$ a session type in the same sequent starting with a corresponding then both actions of both session types are cut out. $(A; end, \overline{A}; end)$ implies (end, end). E.g,

$$\frac{?a; end, end, end}{?a; end, ?b; end, !b; end}\ PREFIX$$

5) Par: $[T\$U]$ becomes the set of two sequents [T],[U] $[T\$U, V]$ becomes $(T \wedge V) \vee (U \wedge V)$. E.g,

$$\frac{\{?a; end, ?b; end\} \quad \{!b; end\}}{?a; end, ?b; end\$!b; end}\ PAR$$

Or,

$$\frac{\{?a; end, !b; end\} \quad \{?b; end\}}{?a; end, ?b; end\$!b; end}\ PAR$$

Use case instances of the algorithms applied on session types will be made in the following technical deliverable.

4.4. Assessment

The scientific deliverable successfully established why and how subtyping relations between session types can be proven by an algorithm. As the main challenges faced with this problem were defined, it lays a perfect ground for implementing this algorithm in order to integrate it to the existing subtyping checking tool.

5. Extension of a Subtyping tool

5.1. Requirements

Here are all the listed requirements of this technical deliverable is :
- [A] Extending the session subtyping tool [1]
- [B] Modify the tool's existing parser
- [C] Developping an Algorithm for checking parallel subtyping
- [D] Developping the algorithm with a proof search structure, searching for the solution autonomously
- [E] Adding the parallel subtyping tool to the Graphical User interface of the subtypying tool

5.2. Design

- [A] The tool's source code is available on github, it can freely be downloaded and extended. The algorithm's used in the tool are implemented in the functional programming language of Haskell. Once compiled, the executable file can be added in a specific folder of the complete tool. Every algorithm present in the tool is defined in a *json* configuration file, in which we define the path of the executable file and the command accompanied with it's parameters to correctly execute the algorithm. The rest of the software (the graphical user interface GUI) was implemented in python, it does not need any modification. Finally, internal communications present in session types will be viewable under the form of an automata.
- [B] The tool is relying on several parser's to handle the user's input. First of all, a parser and linter was generated in python, to verify if the user is respecting the correct syntax of session types. This check takes place before calling the executable algorithm. Afterwards, each algorithm implemented in haskell has it's own parser, which is used to attribute a signature to each type according to type of session type. Since we introduce new symbol and session types to the tool, we will need to modify these two parser's to implement the new algorithm in the existing tool.
- [C] After modifying the Haskell parser present in the source code, we can start by implementing the main algorithm. By using the GUI, two temporary text files are created to store the sub and super type inserted by the user. These files are read by the program, handled according to the algorithm. Finally we obtain the time taken by the algorithm and one of two results, either the subtyping relation holds or it does not. In both cases, the result is shown to the user in a pop out window of the GUI. Furthermore, a temporary result file is created, this file is showing all the steps and intermediate states taken by the algorithm. The user has access to this file by navigating the tool's interface.
- [D] The algorithm which is going to be implemented is applying a set of inference rules leading to a prove that the subtyping relation holds. These choices are non-deterministic, we therefore need to define a structure and set of rules for the algorithm to take, in order for the algorithm to autonomously apply these rules to take a verdict.
- [E] Once the algorithm was implemented, the executable version needs to be integrated to the complete software. The user will be able to execute the algorithm, seing the result and the log text file.

5.3. Production

5.3.1. Original Subtyping tool. As the tool is considered to be easily extensible, it is sufficient to simply modify a *.json* configuration file to add the new algorithm. In addition, the main modification involved the change of the tool's python parser and lexer, the viewer executable allowing us to visualise internal communications, and the change of the file opener to accommodate text files instead of image files, which were used for the existing algorithms. In figure 10 the main interface of the existing subtyping checker tool is visible. To use it, a subtype in the left text field and a supposed supertype on the right text field shall be defined. The original tool does not support any internal communications of session types, as previously explained, we introduced to it the '$|$' symbol acting as internal communication. As well, as the "$" symbol acting as the Dual of the '$|$' symbol. On the top navigation bar, we can chose among 4 algorithms, the first 3 of them, are simulation algorithms creating simulation graphs to verify if the subtyping relation holds. The fourth one, is the algorithm presented in the previous scientific deliverable that is developed as the main technical deliverable of this project. By clicking on the algorithm in the upper navigation bar, we can apply it to the two session types defined below.

In the software, it is possible to view the session types under the form of an automata. Once the parser is modified, we can use session types having internal communications, and view them in the tool's viewer. For example, for the session type : $?a; end|?a; !a; end$, the image file (*.png* format) in figure 7 will be generated. The parser simply seperates the two session types, creates a graph for each, to finally merge them as one image. The graphs are generated with *Graphviz*, an open source graph visualisation software,in a *.dot* format. For every action (sending or receiving), the program creates a state and a relation in the graph. The graph for both session types would be defined as follows in a .dot file syntax:

```
digraph G {
    q1[shape= "rectangle" label= "1"]
    q2[shape= "rectangle" label= "2"]
    q1 -> q2 [label="?a"];
    q3[shape= "rectangle" label= "3"]
    q4[shape= "rectangle" label= "4"]
    q5[shape= "rectangle" label= "5"]
    q3 -> q4 [label="?a"];
    q4 -> q5 [label="!a"];
}
```

The Haskell program acting as the viewer originates from the existing subtyping tool, it was modified to handle internal communications.

5.3.2. Modifying the Parser. A second modification made to the source code is concerning the two parser's used by the tool. For implementing parallel session types in the session subtyping tool we need to modify the lexer and parser verifying the session type syntax. To be generated, the Parser is created from a grammar, it is defined in a *.g4* file. The following grammar is used,

$$G \rightarrow \mu D.G| + D; G, D; G|\&D.G; , D; G|!D; G|?D; G|D|$$

$$end \text{ PRL } G'|D \text{ PRL } G'|end$$

$$G' \rightarrow \mu D.G'| + D; G', D; G'|\&D.G'; , D; G'|!D; G'$$

$$|?D; G'|D|end$$

$$D \rightarrow a..z|A..Z$$

$$PRL \rightarrow' |'|\$$$

$$OP \rightarrow \&|+$$

Using ANTLR, a parser generator, the tokens, a python Parser and Lexer are generated from the grammar file. The following command it used,

```
antlr4 -Dlanguage=Python3 -visitor file.g4
```

Once parsed by the python parser, the parsed text is parsed by a rather limited haskell parser, assigning the text to custom datatypes to be used by the algorithm. A *LocalType* is composed by a series of it's subtypes defined below,

```
Direction = Send | Receive

Seperator = Bar | Dollar

LocalType = Act Direction String LocalType
    | Rec String LocalType
    | Var String
    | End
    | Choice Direction [LocalType]
    | Prl LocalType Seperator LocalType
```

For example, the session type, $?a; end|end$, would be defined as

```
(Prl (Act Receive "a" End) Bar (End))
```

As Haskell is a functional programming language, every function is defined in terms of pattern matching. A function has a definition for every different input pattern. For every case (every different structure), it is decided which definition of this function to apply to the input. For a function taking as an input a LocalType, it will have a different definition depending on its structure. Let's say we want to dualize a type,

```
d :: LocalType -> LocalType
d (Act Send s lt) = (Act Receive s (d lt))
d (Act Receive s lt) = (Act Send s (d lt))
d (Prl lt BAR s) = (Prl (d lt) DOLLAR (d s))
d (Prl lt DOLLAR s) = (Prl (d lt) BAR (d s))
d (End) = End
```

Every function in our program will pattern check the given Session Type, and return a new session type without modifying the input. Using recursion, we can apply a function to the entire session type, since a session type is a series of structured patterns.

5.3.3. Implementation of the Parallel Subtyping Algorithm. As it is presented in the scientific deliverable, once the session type is parsed, depending on the parameter either the supertype or the subtype is dualized, this will prove one of the two definition of sutyping we layed down (1. or 2.). For efficiency purposes, these two types are defined in the datastructure of a Multiset (bag) to represent sequents. A multiset doesn't take into account about the order of elements, e.g., $\{a, b\} = \{b, a\}$; but does keep track of the number of occurrences, e.g., $\{a, a\} \neq \{a\}$." For the subtyping relation to hold, the program applies a set of rule to modify this branch until the *OK* rule can be applied.

For this problem, using proof search, the program is finding all the possible paths, one of them could be the right proof. By trying to prove that subtyping holds, a set of different rules can be applied in a non-deterministic way. It is therefore important to find the right rule and apply them respectively.

The algorithm is applying a set of rules to the given Session types until no more can be applied. Each rule is defined as a function in the program. Applying the knowledge from the scientific deliverable (Curry–Howard correspondence), a function of this program of type (Multiset LocalType) $\rightarrow$ (Multiset LocalType) which is taking as an input a Branch of types and returning a new branch of types, is connoting a proof that the first Branch implies the second branch.

The order in which the asynchronous rules apply does not matter. By applying these rules on a branch, the *TIMES* rule will create one new branch, the *JOIN* rule, however, will create two new branches from the current one. The functions of these rules are considering a branch as the input and returns a list of branches.

The synchronous set of rules rather implies importance on the order in which they are applied. For each order in which these rules are enforced, it is considered as a new tree. Every one of these rules generates several possibilities, for which only one needs to hold. The functions of these rules are considering a branch as the input, and returning a list of trees.

Starting from the initial branch, the asynchronous rules are applied, generating a list of branches. For each of these branches, we reapply if needed, the asynchronous rules before applying the synchronous ones, again and again until there exists no more branch where any rule can be applied. Forthwith we end up with a list of tree's each having a list of branches. For the sake of efficiency, the program only keeps track of the most recent branches (upper-most) in a tree.

Under those circumstances, as the rules are applied, a list of trees is obtained, each tree has a list of it's uppermost branches, each branch is a multiset of session types. Each tree could be considered as a different path to the right proof. As a reminder, for the subtyping relation to hold, there must exist one tree, which has every branch holding with the *OK* rule. Checking this condition is done naively, there is no need to verify every tree if one correct tree is already found.

Let's come back to an example treated in the scientific deliverable, to illustrate a use case instance of this algorithm. Does the below subtyping relation hold ?

$$?a; end \leq !b; ?a; end | ?a; end$$

After dualizing the supertype, no asynchronous rule can be applied. It is finally the *PAR* rule followed by the *PREFIX* rule that is applied. The final data structure generated by the program is the following,

Tree 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{end, end}{?a; end, !a; end}\,PREFIX \qquad end}{?a; end, !a; end \$ end}\,PAR}{?a; end \leq !a; end \$ end}\,FIRSTSTEP}{?a; end \leq ?a; end | end}\,DUAL$$

Tree 2:

$$\dfrac{\dfrac{\dfrac{\dfrac{Blocked}{?a; end, end}\,NONE \qquad \dfrac{Blocked}{!a; end}\,NONE}{?a; end, !a; end \$ end}\,PAR}{?a; end \leq !a; end \$ end}\,FIRSTSTEP}{?a; end \leq ?a; end | end}\,DUAL$$

Both premises in the first tree or the second tree need to succeed for subtyping to hold, thus, subtyping does hold for this relation. Since the first tree holds, the program did not check the second tree. Apart from the main result that can be seen in figure 9, the user can also view the text file generated by the program, indicating the intermediate states of the algorithm. It is indicating the obtained trees, after applying the synchronous and asynchronous rules. For the previous example, the program generated the following text file,

```
Final Result: Subtyping between
'?a; end' and '?a; end | end' holds.
Synchronous rules got applied:
Tree #1:
Branch 1: [end,end] Branch 2: [end]
Tree #2:
Branch 1:[?a; end,end] Branch 2:[!a; end]
Asynchronous rules got applied:
Tree #1: Branch 1:[!a; end $ end,?a; end]
Dualize the supertype/subtype:
[!a; end $ end,?a; end]
?a; end<=?a; end | end
```

5.3.4. Challenge of the Algorithm. By applying the Prefix rule the algorithm needs to make choices in terms of with which element of the Sequence we should start and which dual action to choose from, by taking the right choice we can correctly cancel out the correct dual communications, however wrong choices imply a blocking in the algorithm. When blocked the algorithm cannot pursue on the given branch. Let's take a look at the following example, where a wrong choice implies the blocking of the algorithm, telling us a subtyping relation does not hold even though it does hold.

Tree 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Blocked}{?a; ?b; ?c; end, end, !b; !c; !a; end}\,None}{?a; ?b; ?c; end, ?a; end, !a; !b; !c; !a; end}\,PREFIX}{?a; ?b; ?c; end | ?a; end, !a; !b; !c; !a; end}\,TIMES}{?a; ?b; ?c; end | ?a; end \leq !a; !b; !c; !a; end}\,FIRSTSTEP}{?a; ?b; ?c; end | ?a; end \leq ?a; ?b; ?c; ?a; end}\,DUAL$$

As it can be seen above, while the program applied the *PREFIX* rule, it could have canceled out either the first action in $?a; end$, or the first action in $!a; !b, !c; !a; end$. The first choice makes the algorithm stuck, even though the subtyping relation holds.

A solution to this choice problem would be, to represent every choice of the Prefix rule in a different sequence (branch). For every new branch, a new tree is created. The new tree also needs to include the other branches of the originating tree. By coming back to the previous example, the *PREFIX* rule would create a second tree,

Tree 2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{end, end, end}{end, ?a; end, !a; end}\,PREFIX}{?c; end, ?a; end, !c; !a; end}\,PREFIX}{?b; ?c; end, ?a; end, !b; !c; !a; end}\,PREFIX}{?a; ?b; ?c; end, ?a; end, !a; !b; !c; !a; end}\,PREFIX}{?a; ?b; ?c; end | ?a; end, !a; !b; !c; !a; end}\,TIMES}{?a; ?b; ?c; end | ?a; end \leq !a; !b; !c; !a; end}\,FIRSTSTEP}{?a; ?b; ?c; end | ?a; end \leq ?a; ?b; ?c; ?a; end}\,DUAL$$

As it can be seen in the second tree, making the second choice results in proving that the subtyping relation indeed holds.

Another challenge may be that some Asynchronous rules cannot be applied before Synchronous rules even though, the order is the other way around. The implementation of this algorithm took into account this possible problem occurence. The following example illustrates this use case instance,

$$?a; +\{?a; end, ?b; end\} \leq ?a; ?a; end$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{end, end}{!a; end, ?a; end}\ OK \qquad \dfrac{Blocked}{end, ?a; end}\ NONE}{+\{!a; end, end\}, ?a; end}\ JOIN}{?a; +\{!a; end, end\}, ?a; ?a; end}\ PREFIX}{?a; +\{!a; end, end\} \leq ?a; ?a; end}\ FIRSTSTEP}{?a; +\{!a; end, end\} \leq !a; !a; end}\ DUAL$$

The subtyping relation does not hold, however, the algorithm did not get blocked since it applied Asynchronous rules after applying the synchronous ones. The program goes through a cycle, once both set of rules got applied, it is recognized that more rules can still be applied, the branches go through the cycle again.

5.3.5. Integration of the Algorithm into the existing tool.

Using a *Haskell* compiler, the source code is compiled into an executable binary file which is added to the main tool, in a folder specifically created for this algorithm. Once the tool's configuration file is modified, the user will be able to verify subtyping on session types using this algorithm (by choosing from the algorithms list on the top navigation bar of the tool, figure 8). Furthermore, the new *Viewer* and the new *Parser* are once compiled, added to the existing corresponding folders of the tool. The source code of the algorithm, the *Viewer* and the *Parser* are available on GitHub [8]. In addition, the complete extended tool with the precompiled executables is also available in the same repository.

5.4. Assessment

As stated, the final technical deliverable got integrated into the existing sub-typing checking tool. The user of the Graphical User Interface is able to define session types using internal communications, dualize them and verify if the sub-typing relation between the subtype and it's supposed supertype holds. Once executed, the result and time taken by the algorithm are displayed. By opening the "result" tab on the upper menu bar, the result text file can be opened. The different states of the execution and main steps taken by the algorithms are written down. However, due to time and technical knowledge limitations, not every intermediate state of the algorithm is visible in the text file. Despite working on most cases, the algorithm has some misinterpretations about some edge cases that shall be rectified in future work. In addition, one of the most important aspects of the algorithm, efficiency, can be improved by changing the way the algorithm proceeds to prove sub-typing. Instead of computing every possible tree of the relation to finally check if the condition holds for at least one tree, the condition could be checked once a tree cannot extend anymore. The conditions shall be checked before other trees are uselessly generated. A second way to improve efficiency is too apply heuristics instead of naively generating every possible course of action.

Acknowledgment

The author would like to thank the BiCS management, the education team, and his tutor Ross James Horne for the fantastic help as well as their availability. I would like to thank Lorenzo Bacchiani for publishing his source code and making it extendible.

6. Conclusion

After presenting and explaining an efficient proof system for checking subtyping of session types, a possible implementation and integration of this algorithm which was never implemented before. This implementation allowed us not only to extend an already existing subtyping checker tool but also to make it more powerful by making it compatible with session types having internal communications. Future work could be consisting of perfecting the tool by fixing the issues stated in the Assessment part to finally contribute this extension to this open-source project.

7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:

    1) Not putting quotation marks around a quote from another person's work
    2) Pretending to paraphrase while in fact quoting
    3) Citing incorrectly or incompletely
    4) Failing to cite the source of a quoted or paraphrased work

5) Copying/reproducing sections of another person's work without acknowledging the source
6) Paraphrasing another person's work without acknowledging the source
7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
8) Using another person's unpublished work without attribution and permission ('stealing')
9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Lorenzo Bacchiani. A session Subtyping Tool (Extended Version) University of Bologna
http://arxiv-export-lb.library.cornell.edu/pdf/2104.12455

[2] Mariangiol Dezani Ciancaglini and Ugode'Liguoro. Sessions and Session Types : An Overview.
www.researchgate.net/publication/225835882_Sessions_and_Session_Types_An_Overview

[3] Mario Bravetti. A SOUND ALGORITHM FOR ASYNCHRONOUS SESSION SUBTYPING AND ITS IMPLEMENTATION. https://arxiv.org/pdf/1907.00421v5.pdf

[4] Marco Carbone. FRIDA2020 Conference, An Introduction to Session Types.
https://www.youtube.com/watch?v=Qnf845kHC3I

[5] K. Honda. Types for dyadic interaction.
https://link-springer-com.proxy.bnl.lu/chapter/10.1007/3-540-57208-2_35?pds=4102021103046191410404071827918962

[6] Ross James Horne. Session Subtyping and Multiparty Compatibility using Circular Sequents.
https://drops.dagstuhl.de/opus/volltexte/2020/12824/pdf/LIPIcs-CONCUR-2020-12.pdf

[7] MIT Licence, of the Session-Subtyping-Tool developed by Lorenzo Bacchiani.
https://github.com/LBacchiani/session-subtyping-tool/blob/main/LICENSE

[8] Source Code of the Session-Subtyping-Tool extension.
https://github.com/jetlime/Subtyping_Session_Tool_SourceCode

[9] Philip Wadler, Propositions as types.
https://dl.acm.org/doi/fullHtml/10.1145/2699407

8. Appendix

All images used in this report.

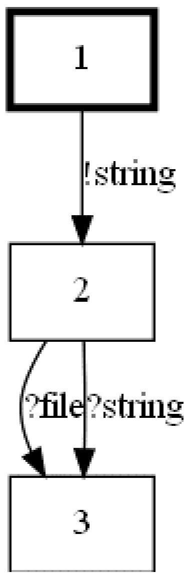


Figure 1. Visual representation of a session type defined in section *4.3.1*

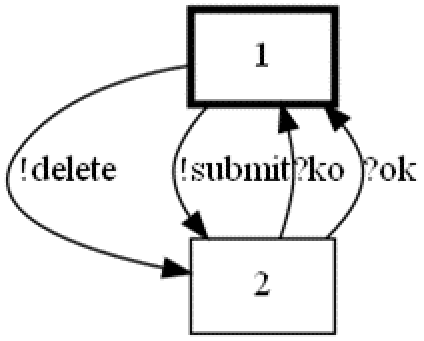


Figure 2. First example of section *4.3.2*, User Session type

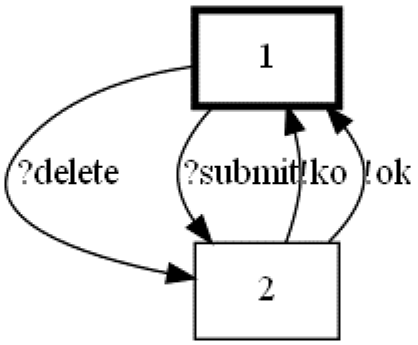


Figure 3. First example of section *4.3.2*, Server Session type

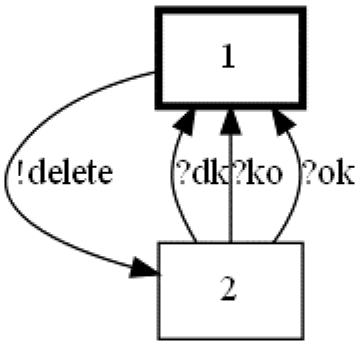


Figure 4. Section example of section *4.3.2*, USER' session type

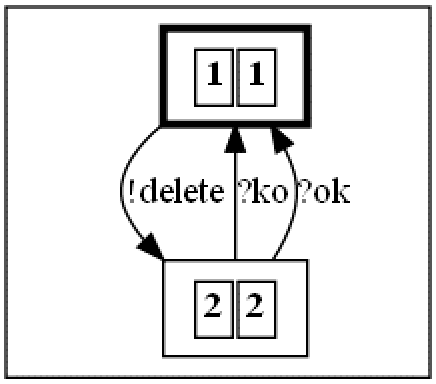


Figure 5. Synchronous subtyping simulation on the on the supposed $USER' \leq USER$ relation

[OK] $[\Theta]\ \mathsf{ok}\ ,\ \mathsf{ok}\ ,\ \ldots\ \mathsf{ok} \vdash$

[LEAF] $[\Theta \mathbin{]\![} \Gamma]\ \Gamma \vdash$

[FIX-μ]
$$\frac{[\Theta \mathbin{]\![} \mu \mathbf{t}.\mathsf{T}\ ,\ \Gamma]\ \mathsf{T}\{^{\mu \mathbf{t}.\mathsf{T}}/_{\mathbf{t}}\}\ ,\ \Gamma \vdash}{[\Theta]\ \mu \mathbf{t}.\mathsf{T}\ ,\ \Gamma \vdash}$$

[MEET]
$$\frac{[\Theta]\ ?\lambda_j;\mathsf{T}_j\ ,\ \Gamma \vdash \quad \text{for } \textbf{some } j \in I}{[\Theta]\ \bigwedge_{i \in I} ?\lambda_j;\mathsf{T}_i\ ,\ \Gamma \vdash}$$

[JOIN]
$$\frac{[\Theta]\ !\lambda_j;\mathsf{T}_j\ ,\ \Gamma \vdash \quad \text{for } \textbf{all } j \in I}{[\Theta]\ \bigvee_{i \in I} !\lambda_j;\mathsf{T}_i\ ,\ \Gamma \vdash}$$

[PREFIX]
$$\frac{[\Theta]\ \mathsf{T}\ ,\ \mathsf{U}\ ,\ \Gamma \vdash}{[\Theta]\ !\lambda;\mathsf{T}\ ,\ ?\lambda;\mathsf{U}\ ,\ \Gamma \vdash}$$

[TIMES]
$$\frac{[\Theta]\ \mathsf{T}\ ,\ \mathsf{U}\ ,\ \Gamma \vdash}{[\Theta]\ \mathsf{T} \otimes \mathsf{U}\ ,\ \Gamma \vdash}$$

[PAR]
$$\frac{[\Theta]\ \mathsf{T}\ ,\ \Gamma_1 \vdash \quad [\Theta]\ \mathsf{U}\ ,\ \Gamma_2 \vdash}{[\Theta]\ \mathsf{T} \parr \mathsf{U}\ ,\ \Gamma_1\ ,\ \Gamma_2 \vdash}$$

Figure 6. A presentation of the algorithmic coinductive proof system Session [6]

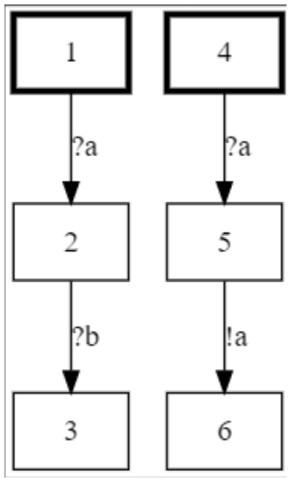


Figure 7. A visual representation of Session Types with internal communications

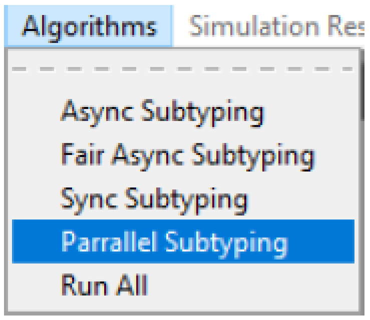
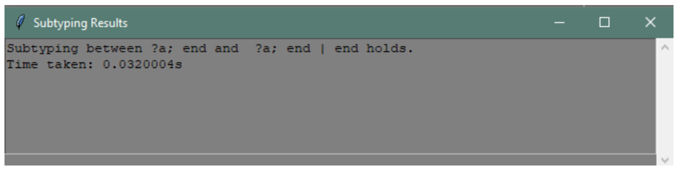


Figure 8. List of algorithms in the extended tool



```
Subtyping between ?a; end and  ?a; end | end holds.
Time taken: 0.0320004s
```

Figure 9. Pop up message indicating the result of the algorithm [1]

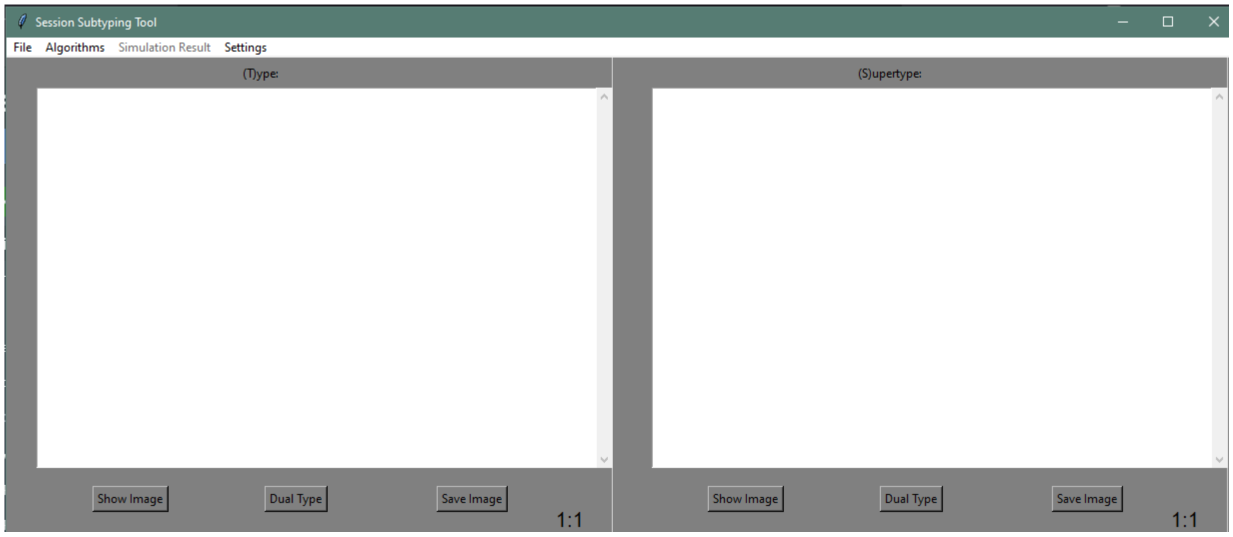


Figure 10. Graphical User Interface of the Subtyping Checker tool [1]