A Type System for the Jolie Language

Julie Meinicke Nielsen

Kongens Lyngby 2013
IMM-M.Sc.-2013-0074

Supervised by

PhD Student
Fabrizio Montesi
The IT University of Copenhagen

Associate Professor
Marco Carbone
The IT University of Copenhagen

Associate Professor
Nicola Dragoni
Technical University of Denmark

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-0074

Summary UK

The Jolie Language is a general-purpose service-oriented programming language. Service-oriented programming languages are designed for writing applications to services in. A service is an entity in a loosely coupled distributed system which is structured after the Service-Oriented Computing (SOC) paradigm. The focus of SOC is to separate software engineering from application programming and therefore are services autonomous entities which communicate by exchanging messages [MGZ, TCBM06].

In Jolie messages are structured as trees. A variable in Jolie is a path in a data tree and the type of a data tree is a tree itself. Jolie provides a language for describing the types that are allowed to be communicated in a network. Communications are type checked at run-time when a message is received, and such type checking is not formally defined. The aim of this thesis is to introduce static type checking to the theoretical foundation of the core fragment of Jolie.

The way of structuring types and handling variables in Jolie creates some challenges for introducing type checking to Jolie: The type language provided by Jolie allows a subtree of a tree type to be optional. Equality of types must therefore be handled with that in mind. Variables are not declared wherefore the manipulation of the program state must be inferred. Besides the design of Jolie, SOC itself is also challenging: The ability to specify the type of messages a service wants to receive does not prevent that an ill-formed service send a message with a wrong type. It is therefore necessary to formalize run-time type checking of incoming messages.

The contribution of this thesis is the design of a static type checker for the core fragment of Jolie. The type checker rejects networks of services in which

a message is sent or received, where the message has a wrong type according to sender and receivers type specifications. This is formally proved along with the property that a well-typed network can not reduce to an ill-typed network. Furthermore the dynamic type checking of incoming messages is described formally.

Summary DK

Jolie er et general-purpose service-oriented programmeringssprog. Service-oriented programmeringssprog er lavet til at skrive programmer til services i. En service er en enhed i et løst koblet distribueret system, som er struktureret efter paradigmet, Service-Oriented Computing (SOC). Fokus i SOC er at separere software engineering fra applikationsprogrammering, og derfor er services autonome enheder som kommunikerer ved at udveksle beskeder [MGZ, TCBM06].

I Jolie er beskeder struktureret som træer. En variabel i Jolie er en sti i et datatræ, og et datatræs type er selv et træ. Jolie tilbyder et sprog til at beskrive de typer der må kommunikeres i et netværk. Kommunikation bliver typetjekket under kørslen når en besked bliver modtaget, og dette er ikke formelt defineret. Formålet med dette speciale er at introducere statisk typetjek til den teoretiske understøttelse af kernefragmentet af Jolie.

Den måde typer er struktureret og variabler er håndteret på i Jolie, skaber nogle udfordringer med hensyn til at indføre typetjek i Jolie: Jolies typesprog tillader at et eller flere deltræer i en type ikke er obligatorisk for typen. Dette må tages i betragtning ved håndtering af lighed imellem typer. Eftersom variabler ikke erklæres skal ændringerne i programtilstanden udledes. Udover designet af Jolie giver også selve begrebet SOC anledning til udfordring: Muligheden for at specificere typen af beskeder som en service ønsker at modtage, forhindrer ikke at en dårligt formuleret service sender en besked med en forkert type. Det er derfor nødvendigt at formalisere run-time typetjek af indkomne beskeder.

Dette speciale bidrager med et design af en statisk typetjekker til Jolies kernefragment. Typetjekkeren afviser netværk af services hvori der sendes eller mod-

tages beskeder med en forkert type i forhold til afsender og modtagers type-specifikationer. Dette bevises formelt sammen med den egenskab at et well-typed netværk ikke kan reducere til et ill-typed netværk. Derudover beskrives dynamisk typetjek af indkomne beskeder formelt.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis introduces type checking to the theoretical foundation of the core fragment of the Jolie language, which excludes recursive types, arrays, subtyping of basic types, faults and deployment instructions such as architectural primitives. We describe the core fragment of Jolie in chapter 1, while the complete Jolie language is described in [Jol]. The Jolie Language is a general-purpose service-oriented programming language. Service-oriented programming languages are designed for writing applications to services in. A service is an entity in a loosely coupled distributed system which is structured after the Service-Oriented Computing (SOC) paradigm. The focus of SOC is to separate software engineering from application programming and therefore are services autonomous entities which communicate by exchanging messages [MGZ, TCBM06]. In order to support the web services specifications[W3C] Jolie allows data to be structured in trees where the edges corresponds to e.g. XML tags and the nodes corresponds to the values of the tags. Jolie provides a language for describing the types of its messages. In this language a subtree of a tree type is allowed to be optional. Equality of types must therefore be handled with that in mind. As a solution we introduce subtyping in 2.3.

The type language is used to declare types for messages allowed to be communicated through a specific operation for communication [Jol]. Variables are not declared and the manipulation of the program state must therefore be inferred. We handled this in the type system which is presented in 3.2.

Besides the challenges from the way of structuring data and handling variables, the service-oriented computing itself is also challenging: The ability to specify the type of messages a service wants to receive using an operation does not prevent that an ill-formed service send a message with a wrong type. It is therefore necessary to formalize run-time type checking of incoming messages. We extend the semantics of Jolie with type checking of incoming messages in 2.4.

A static type checker for checking communication will reveal the communication type errors in a network at compile time. Type errors can therefore be catched by the programmer instead of by the user. This is a benefit since the programmer can handle errors beforehand, and since type errors might first show up when the service is used by many other services. The formally defined type checking of communication will create the foundation of type checking networks since it addresses the interaction between services.

This thesis tackles the problem of detecting type errors in service-oriented systems by the use of the Jolie language.

The contribution of this thesis is the design of a static type checker for the core fragment of Jolie. The type checker rejects networks of services in which a message is sent or received, where the message has a wrong type according to sender and receivers type specifications. This is formally proved along with the property that a well-typed network can not reduce to an ill-typed network. Furthermore the dynamic type checking of incoming messages is described formally.

The Jolie language is constructed in three layers: The behavioural layer deals with the internal actions of a process and the communication it performs seen from the process's point if view, the service layer deals with the underlying architectural instructions and the network layer deals with connecting communicating services [Mon10]. The thesis follows the layered structure of Jolie when presenting the syntax and semantics of the Jolie language, the type system and the properties type preservation and type safety:

Chapter 1 introduces the core fragment of Jolie and presents its syntax. The tree structure of variables in Jolie is explained and the structure of their types are presented.

Chapter 2 presents the semantics of Jolie. The dynamic type checking of incoming messages is introduced together with subtyping which it makes use of.

Chapter 3 presents the typing relations and the type system. Furthermore the

type preservation and type safety properties are presented and proved.

Chapter 4 discuss this work and future work.

The appendix A contains the semantics and typing rules presented in this thesis.

Lyngby, 15-July-2013

Julie Meinicke Nielsen

Acknowledgements

I would like to thank my supervisors for the interesting discussions we have had, and for answering my long lists of questions. I would like to give a special thank to Fabrizio Montesi for presenting me such an interesting master thesis topic and for being so enthusiastic that even after having worked with this topic in half a year I still enjoy the work very much.

I would also like to thank the people in The Programming, Logic and Semantics Research Department at The IT University for treating me like I was one of you, and thereby making me feel very welcome. My special thanks go to Ornela Dardha and Marco Paviotti for interesting discussions about process calculi, type systems and session types. Second thanks go to Marco Paviotti for letting me borrow a computer when my own broke down a month before deadline.

Finally, I would like to thank my parents for giving births to me and not regretting it.

Contents

Chapter 1

The Jolie Language

This chapter provides the necessary background information on the core fragment of the programming language Jolie (Java Orchestration Language Interpreter Engine). It is reported from [MGZ, Mon10, MC11, Jol].

Jolie is a service-oriented programming language. A Jolie program consists of two part: The behavioural part defines a service behaviour, and the deployment part defines the composition of the service with the rest of the network.

Jolie is build on SOCK [GLG+06] which is a process calculus for modelling service-oriented systems. This is done by having three layers: The internal computations of a process and the communication projected to the process is specified at the behavioural layer, while the underlying description of the communication, architecture and state of the process is described at the service layer. Communication in a network is described at the network layer [Mon10, MGZ].

1.1 Behavioural Layer

The behavioural layer describes the internal actions of a process and the communications it performs seen from the process' point of view.

The statements at the behavioural layer are called behaviours and they are ranged over by B. Expressions are ranged over by e, channel names are ranged over by r, operation names are ranged over by o, locations are ranged over by l and variables are ranged over by x. In this thesis we consider the behaviours described by the following grammar:

$$
\begin{array}{rll}
B ::= & \eta & (input) \\
| & \overline{\eta} & (output) \\
| & \texttt{if(}e\texttt{)}\ B_1[\texttt{else}\ B_2] & (if) \\
| & \texttt{while(}e\texttt{)}\ \{B\} & (while) \\
| & B_1; B_2 & (sequence) \\
| & B_1 \mid B_2 & (parallel) \\
| & \texttt{x = }e & (assign) \\
| & \mathbf{0} & (nil) \\
| & [\eta_1]\{B_1\} \cdots [\eta_n]\{B_n\} & (input\ choice) \\
| & \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) & (wait) \\
| & \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B) & (exec) \\
\eta ::= & \texttt{o(x)} & (one-way) \\
| & \texttt{o(x)(x')\{}B\texttt{\}} & (request-response) \\
\overline{\eta} ::= & \texttt{o@}l\texttt{(e)} & (notification) \\
| & \texttt{o@}l\texttt{(e)(x)} & (solicit-response)
\end{array}
$$

Communication is available through rules (input), (output) and (input choice). Input communication can either be unidirectional (one-way) or bidirectional (request-response). Both stores the input received on operation o in variable x, but request-response in addition executes behaviour B and reply with the content of x'. The three actions are done in sequence of request-response are done in sequence.

The rules for output communication, (notification) and (solicit-response) are the dual of respectively (one-way) and (request-response). The output communication rules differs from the input communication rules in that they have an extra parameter. The location l of the receiving service can e.g. be represented by an URL. In the full Jolie syntax the extra parameter is an output port instead of a location. The communication port is specified at the service layer and it consists of location, protocol, interface and architecture settings. The communication port is not essential for the purpose of designing a type system, hence it is omitted. For a detailed description of communication ports see [MGZ].

Each option in an (input choice) consists of a guard η which ranges over input options and a behaviour B, which is executed if the guard allows it.

The behaviours B-Exec and B-Wait are runtime statements. They were included in the Jolie language in [MC11]. The versions of them presented here differ from their versions in [MC11] in the way that they also have the operation as parameter. We added it since it is needed in section 3.3.4 where we prove the preservation property for the typing system.

The behaviours (parallel) and (nil) are common concurrent statements. The first is a construct of two behaviours running in parallel, while the latter is a construct of doing nothing.

The behaviours (while), (if) and (sequence) are common statements in imperative languages. The brackets around the else-branch in (if) indicates that the branch is optional.

Since variables in Jolie have tree structure, the (assign) behaviour works different from common assign behaviours. A variable in Jolie is a path in the variable tree which it is a part of. The subtree of a path consists of a value (node), and optional continuations of the path (edges). The (assign) rule assigns a value to a path without influencing its possible children.

1.1.1 Jolie variables

In Jolie the data structure for a variable is a tree, where the nodes contain values. Consider for instance the variable named amount, which gets assigned the value 12:

```
amount = 12
```

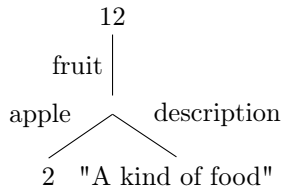
It is represented as a tree with a single node:

12

A variable can be extended by adding edges to it delimited by a ".":

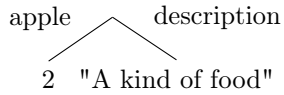
```
amount.fruit.apple = 2;
amount.fruit.description = "A kind of food"
```

The tree for amount now got extended:



Note that a node is allowed not to have any value at all.

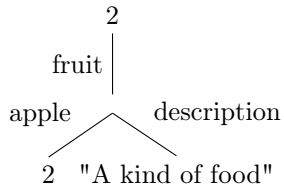
A variable is a path in a data tree. For instance the data tree for variable amount.fruit is:



As described in section 1.1 the assign statement does not changes the children of the variable it is used at:

```
amount = 2
```

The tree for amount:



1.1.2 Types

A type is a tree, where the nodes contain basic types. A basic type BT can be one of the following:

$BT ::=$ `bool` | `int` | `double` | `long` | `string` | `raw` | `void`

Type `raw` is used for data streams and type `void` is used for no data. The rest are standard. The full Jolie syntax also includes the type `any` which is used for a basic type which can be any of the basic types.

Jolie trees are ranged over by T. A Jolie tree consists of a basic type and an optional list of children, CTL:

$$T ::= BT \mid BT\,\{CTL\}$$

A list of children consists of a child, CT, and a list of children:

$$CTL ::= CT\,CTL$$

Notice, that a list of children can not be empty. A child of a type tree consists of an id, a cardinality and a type tree:

$$CT ::= .id\,C : T$$

The cardinality specify a range over allowed number of array elements. In this fragment of Jolie the following ranges are considered:

$$C ::= [0,0] \mid [0,1] \mid [1,1] \mid [?]$$

The last production is a shortcut for $[0,1]$. Not specifying a cardinality is a shortcut for $[1,1]$, which is the only cardinality allowed for root nodes.

1.2 Service Layer

The service layer contains the structures for handling communication and program run. The entities at the service layer are processes and services.

Besides the behaviour a process also consists of a state t and a message queue $\tilde{m}$:

$$P ::= \quad B \cdot t \cdot \tilde{m} \quad | \quad \mathbf{0} \quad | \quad P \mid P$$

We let P range over processes, t range over trees and $\tilde{m}$ range over message queues. A state is a tree which consists of all the variables in the process, such that each variable in the process is represented by a path in the state. A message queue is a FIFO queue of incoming messages. A message is ranged over by m and has the form (r, o, t), where r is the channel it is received on, o is the operation it is received as part of and t is the content of the message.

A $\mathbf{0}$ process is a process that do nothing, has no state and no message queue. An important semantic difference between a $\mathbf{0}$ process and a process $\mathbf{0} \cdot t_{\perp} \cdot \epsilon$ is that a $\mathbf{0}$ process can not receive messages.

A process is part of a service S. Besides processes a service engine also consists of a deployment D and a behaviour:

$$S ::= \quad B \triangleright_D P$$

A service can have at least one of the roles client and server. The behaviour of a service in the role of a server is (input choice). The initial form of a service solemnly in the role of a server is:

$$\sum_{i \in J} [\eta_i]\{B_i\} \triangleright_D \mathbf{0}$$

When processing incoming requests it will spawn processes and run them in parallel with $\mathbf{0}$.

The behaviour of a service solemnly in the role of a client is (nil). The service has the form:

$$\mathbf{0} \triangleright_D B \cdot t \cdot \tilde{m}$$

The deployment denoted D contains information about correlation sets for processes running by the service and type declarations for operations known by the service. In [MC11] only the information about correlation sets are known by

a service. We added a typing environment with declarations for operations in order to be able to extend the Jolie semantics with a dynamic type checker. A deployment is of the following form:

$$D = \alpha_C \cdot \Gamma$$

A correlation set, c is a set of variables which values are used to identify a process. The aliasing function α_C is used to extract information about where in a message the correlation values for an operation can be found. Correlation sets are outside the scope of this project. For a detailed description of the concept see [MC11].

The deployment D consists of type declarations for operations. We assume that the proceeding of the deployment contains adding the operation type declarations to the typing environment. A typing environment, Γ is a set consisting of type bindings for variables and operations. While the operation type bindings are declared in the deployment part, the variable type bindings are inferred during type checking of the behavioural part. A type environment has the following form:

$$\begin{aligned}
\Gamma ::= \quad & \texttt{o@}l : \texttt{<}T\texttt{>} \\
| \quad & \texttt{o@}l : \texttt{<}T, T\texttt{>} \\
| \quad & \texttt{o} : \texttt{<}T\texttt{>} \\
| \quad & \texttt{o} : \texttt{<}T, T\texttt{>} \\
| \quad & \emptyset \\
| \quad & \Gamma, \Gamma \\
| \quad & \cdots
\end{aligned}$$

The structure of an operation type binding is an operation name followed by either a type (unidirectional communication) or a pair of types (bidirectional communication). The production Γ, Γ denotes disjoint union of two environments, while the $\cdots$ denotes that one or more productions are omitted. The omitted production is the form of a variable type declaration. We will explain it in chapter 3. A type declaration for an operation has the form ***key*** : <*type*>. For input communication the key is the operation name, while it for output communication consists of both the operation name and a location of the hosting service. The type is a single type tree for unidirectional communication and a pair of type trees for bidirectional communication.

1.3 Network Layer

The network layer describes the structure of a network. A network denoted N is one or more services running in parallel. We assume that each service is deployed on a unique location. A service S deployed on a location l is denoted $[S]_l$ and the grammar for N is given below.

$$N ::= \quad [S]_l \quad | \quad \boldsymbol{\nu} r\, N \quad | \quad N \mid N \quad | \quad \mathbf{0}$$

We kept the **0** production and additional structural congruence rules (presented in 2.4) from [MC11] even though a service can not be or become **0**, because we otherwise had to make two versions of some of the semantic network rules, in order to make them work for a network consisting of a single service.

Chapter 2

Jolie Semantics

This chapter explains the semantics of Jolie, which is described in a labelled transition system. The labels are described in 2.1 and the semantics is presented in 2.4. We have extended the semantics with a dynamic type check of incoming messages. The dynamic type check is described in 2.2. It makes use of subtyping which is described in 2.3. Except for section 2.2 and 2.3, the semantics of Jolie are reported from [MC11] unless anything else is written. The semantic rules are reported also in a single table in Appendix A, for reference.

2.1 Labels

The semantics of Jolie are described in a labelled transition system, where μ ranges over label names. A transition between behaviours can have one of the following labels:

$$
\begin{array}{rl}
\mu ::= & \texttt{x} = e \\
\mid & \textsf{read}\ t \\
\mid & \boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)} \\
\mid & r : \texttt{o(x)} \\
\mid & (r, \texttt{o})!\, x \\
\mid & (r, \texttt{o@}l)?x \\
\mid & \cdots
\end{array}
$$

Label $\texttt{x} = e$ denotes the action of evaluating expression e and assign the result to variable x. Label $\textsf{read}\ t$ denotes reading the state t of the underlying process. Label $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ denotes sending a expression e through an operation o to a location l over a fresh channel r. Its dual label $r : \texttt{o(x)}$ denotes storing in a variable x a message received through an operation o over a channel r. This label pair is used in unidirectional communication and in first part of bidirectional communication. In the second part of bidirectional communication is the label pair, $(r, \texttt{o})!\, x$ and $(r, \texttt{o@}l)?x$ used. Label $(r, \texttt{o})!\, x$ denotes writing a variable x at channel r as part of an operation o, while $(r, \texttt{o@}l)?x$ denotes reading a message from channel r as part of an operation o and storing it in a variable x. The three dots denotes that the presented productions don't compose the whole grammar for μ. The remaining productions are used at transitions between services:

$$
\begin{array}{rl}
\mu ::= & \cdots \\
\mid & \tau \\
\mid & \boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)} \\
\mid & \boldsymbol{\nu} r\ \texttt{o(}t\texttt{)} \\
\mid & (r, \texttt{o})!\, t \\
\mid & (r, \texttt{o@}l)?t
\end{array}
$$

Label τ represents an internal action. Label $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)}$ and $\boldsymbol{\nu} r\ \texttt{o(}t\texttt{)}$ corresponds to respectively $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ and $r : \texttt{o(x)}$. The difference is that t in $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)}$ denotes the tree of the result of the evaluation of expression e in $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$, while t in $\boldsymbol{\nu} r\ \texttt{o(}t\texttt{)}$ denotes the received message which is going to be stored in variable x in $r : \texttt{o(x)}$.

Likewise label $(r, \texttt{o})!\, t$ and $(r, \texttt{o@}l)?t$ corresponds to respectively $(r, \texttt{o})!\, x$ and $(r, \texttt{o@}l)?x$. These two pairs of labels differs from their version in [MC11]. The difference is that the name for the current operation is added as a parameter. For receiving from an operation the location of the sender is added as well. This

is done because it is needed in the function *sideEffect* and in the semantic rules for transitions labeled `error`. The function *sideEffect* updates the typing environment with respect to the action denoted by the label. It will be introduced in section 3.3.3. The semantic rules for transitions labelled `error` are used in the proof for the progress property. They will be introduced in section 3.4.1.

2.2 Dynamic Type Check

We have extended the semantic rules at the service layer for receiving a message with a dynamic type check. The rules affected are S-Corr and S-Start which is described in 2.4. Recall that a deployment D consists of an aliasing function α_C and an environment Γ, where α_C is used to extract information about where in a message the correlation values for an operation can be found, and Γ contains type bindings for operations known by the service. This is denoted by $D = \alpha_C \cdot \Gamma$. The dynamic type check compares the type of the input operation with the type of the message:

$$D = \alpha_C \cdot \Gamma \quad a(\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma \vee \texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma) \quad \vdash t : T_t \quad T_t \leq T_i$$

where t' is a message sent through operation `o`.

The minimal type T of tree t can be assigned to t with $\vdash t : T$:

Definition 2.1 We write $\vdash t : T$ whenever t is a tree and T is a type such that t has type T and there doesn't exist a type T' such that $T' < T$ and t has type T'.

The type of the message can be determined without use of an environment: Since the value of each node in a tree is copied to the tree instead of copying the pointer, non of the data trees in Jolie contains links, and therefore the type of a data tree can be determined if the types of its nodes can be determined.

Both definition 2.1 and the dynamic type check makes use of subtyping. Definition 2.1 uses it to find the minimal type tree of a data tree and the dynamic type check uses it to compare the type of the input operation with the type of the received message. We describe subtyping below.

2.3 Subtyping

The dynamic and the static type checking make use of subtyping after the principles that a type is a subtype of another type if it is described by the other type. Consider for instance the type t0:

```
type t0 : int { .x:string .y:bool }
```

It is a subtype of t1:

```
type t1 : int { .x[0,1]:string .y:bool }
```

So is t2:

```
type t2 : int { .y:bool }
```

Furthermore each type is a subtype of itself.

We use $\leq$ as the subtyping relation, and we have described this relation with some typing rules. We assume that the abstract syntax trees which the typing rules are applied at, are optimized such that the cardinality $[0, 0]$ doesn't occur in any part of the trees, and such that cardinality shortcuts are written completely.

The subtyping relation between trees is described by ST-T.

$$(\texttt{ST-T}) \quad \frac{BT_1 \leq BT_2 \quad CTL_1 \leq CTL_2}{BT_1\{CTL_1\} \leq BT_2\{CTL_2\}}$$

A tree is a subtype of another tree if its basic type is a subtype of the other trees basic type and if its list of children are a subtype of the other tree's list of children. The subtyping relation between basic types is described by ST-BT.

$$(\texttt{ST-BT}) \quad \frac{BT_1 = BT_2}{BT_1 \leq BT_2}$$

The subtyping relation between lists of children is described by ST-CTL.

$$(\texttt{ST-CTL}) \quad \frac{\begin{array}{c} dom(CTL_1) \subseteq dom(CTL_2) \\ \forall x \in dom(CTL_2)\,.\, CTL_2(x) =< C_2, T_2 > \wedge \\ CTL_1(x, T_2) =< C_1, T_1 > \wedge C_1 \leq C_2 \wedge T_1 \leq T_2 \end{array}}{CTL_1 \leq CTL_2}$$

The first premise of ST-CTL ensures that CTL_1 doesn't have any children which CTL_2 doesn't have. The second premise extract information about the cardinality and tree of each child in the two lists and apply the subtyping relation on these retrievals. The information of a child in list CTL_2 is extracted using the function

$$CTL(x) = \begin{cases} < C, T > & \text{if } x\, C : T \in CTL \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.1}$$

An id x applied on a list CTL returns the cardinality and tree type of the element with id x. It is not defined if x is not an element in CTL. In the premise the function is only used in a space where it is defined, because of the form of the forall condition.

The information of a child in list CTL_1 in the second premise is extracted using the function

$$CTL(x, T_d) = \begin{cases} < C, T > & \text{if } x\, C : T \in CTL \\ < [0,0], T_d > & \text{if } x \notin CTL \end{cases} \tag{2.2}$$

It is defined as the previous function except that it takes as argument a default type tree which it returns together with cardinality $[0, 0]$ in case it is called with an id which is not in the list of children. That way e.g. $t2 \leq t1$ from the above example is typable.

The subtyping relation between cardinalities is described by ST-CTL.

$$(\texttt{ST-C}) \quad \frac{MIN_2 \leq MIN_1 \quad MAX_1 \leq MAX_2}{[MIN_1, MAX_1] \leq [MIN_2, MAX_2]}$$

A cardinality is a subtype of another cardinality if its range is contained in the range of the other cardinality.

The rule ST-T doesn't catch all trees. The rule ST-BT-T catches the case where the subtype doesn't have a list of children. ST-BT-T is a rewriting of ST-T and ST-CTL. Another solution is to view the missing list of children as an empty list. This solution is not chosen because it will create ambiguity since make $BT_1 \leq BT_2$ typable by both ST-T and ST-BT.

$$(\texttt{ST-BT-T}) \quad \frac{BT_1 \leq BT_2 \quad \forall x \in dom(CTL) \,.\, CTL(x) = <C,T> \wedge [0,0] \leq C}{BT_1 \leq BT_2 \{ CTL \}}$$

The case where the subtype doesn't have a list of children but the subtype do is not considered, because of the assumption that all nodes with cardinality $[0, 0]$ are removed.

2.4 Semantics Rules

Recall that Jolie is structured in a behavioural layer, a service layer and a network layer. In this section we present the semantic rules for the statements in Jolie ordered after the layers.

2.4.1 Behavioural Layer

The semantic rules for behavioural statements are described below.

Structural Congruence

$$(\texttt{B-Struct}) \quad \frac{B_1 \equiv B_2 \quad B_2 \xrightarrow{\mu} B_2' \quad B_1' \equiv B_2'}{B_1 \xrightarrow{\mu} B_1'}$$

where structural congruence is defined as follows:

Definition 2.2 (Structural Congruence Rules at Behavioural Layer)

$$\begin{aligned} (\, B_1 \mid B_2 \,) \mid B_3 &\equiv B_1 \mid (\, B_2 \mid B_3 \,) \\ 0; B &\equiv B \\ B_1 \mid B_2 &\equiv B_2 \mid B_1 \\ B \mid 0 &\equiv B \end{aligned}$$

Branches The premises of the semantic rules for the statements (if) and (while) uses notation which requires explanation before presenting the rules:

The function $x(t)$ takes a variable path x and a data tree t and returns the subtree of x in t. It is formally defined in [MC11, p. 5] as:

Definition 2.3

$$x(t) = \begin{cases} t & \text{if } x = \epsilon \\ x'(t') & \text{if } x = a.x' \text{ and } a \text{ is an edge from the root of } t \text{ to } t\text{'s subtree } t' \\ t_\perp & \text{if } x = a.x' \text{ and there is no edge } a \text{ from } t \text{ to a subtree } t' \end{cases} \tag{2.3}$$

where ϵ denotes the empty sequence and $t_\perp$ a tree with a single node with undefined value.

Since the variables of a process are represented as paths in the state of the process, the function $x(t)$ can be used to look up variables in the state. This is used in the function which evaluates an expression on its state:

Definition 2.4 We write $e(t) = t'$ where e is an expression, t is a state and t' is a tree such that t' is the result of the evaluation of e in which each variable in e are looked up in the state t.

$$\text{(B-If-Then)} \quad \frac{e(t)=true}{\texttt{if}(e)\ B_1\ \texttt{else}\ \ B_2 \xrightarrow{\textbf{read}\ t} B_1}$$

$$\text{(B-If-Else)} \quad \frac{e(t)=false}{\texttt{if}(e)\ B_1\ \texttt{else}\ \ B_2 \xrightarrow{\textbf{read}\ t} B_2}$$

$$\text{(B-Iteration)} \quad \frac{e(t)=true}{\texttt{while}(e)\ \{B\} \xrightarrow{\textbf{read}\ t} B;\texttt{while}(e)\ \{B\}}$$

$$\text{(B-No-Iteration)} \quad \frac{e(t)=false}{\texttt{while}(e)\ \{B\} \xrightarrow{\textbf{read}\ t} \mathbf{0}}$$

The semantic rules for the while and if statements are standard. The premise requires evalution of the loop condition given the state read in the transition.

$$\text{(B-Choice)} \quad \frac{j \in J \quad \eta_j \xrightarrow{\mu} B'_j}{\sum_{i \in J} [\eta_i]\{B_i\} \xrightarrow{\mu} B'_j;B_j}$$

Rule B-Choice describes what happens when a guard is taken. The guard is executed in sequence with the body of the chosen branch. In the premise the guard is evaluated one step. This is done because the called service needs to synchronize with the caller service by the label. That way the choice of branch is made by the caller. The synchronization is described in section 2.4.3.

Parallel and Sequence

$$\text{(B-Seq)}\quad \frac{B_1 \xrightarrow{\mu} B_1'}{B_1;B_2 \xrightarrow{\mu} B_1';B_2} \qquad \text{(B-Par)}\quad \frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \mid B_2 \xrightarrow{\mu} B_1' \mid B_2}$$

The semantic rules for the parallel and sequence statements are standard.

Assignment

$$\text{(B-Assign)}\quad \texttt{x} = e \xrightarrow{\texttt{x} = e} \mathbf{0}$$

The assignment action is performed in the transition.

Communication

$$\begin{aligned}
\text{(B-SolResp)}\quad & \texttt{o@}l\texttt{(e)(x)} \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e)}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \\
\text{(B-Notification)}\quad & \texttt{o@}l\texttt{(e)} \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e)}} \mathbf{0} \\
\text{(B-ReqResp)}\quad & \texttt{o(x)(x')\{}B\texttt{\}} \xrightarrow{r:\texttt{o(x)}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B) \\
\text{(B-OneWay)}\quad & \texttt{o(x)} \xrightarrow{r:\texttt{o(x)}} \mathbf{0} \\
\text{(B-Wait)}\quad & \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \xrightarrow{(r,\texttt{o@}l)?x} \mathbf{0} \\
\text{(B-Exec)}\quad & \frac{B \xrightarrow{\mu} B'}{\mathsf{Exec}(r,\texttt{o},\texttt{x},B) \xrightarrow{\mu} \mathsf{Exec}(r,\texttt{o},\texttt{x},B')} \\
\text{(B-End-Exec)}\quad & \mathsf{Exec}(r, \texttt{o}, \texttt{x}, \mathbf{0}) \xrightarrow{(r,\texttt{o})!\,x} \mathbf{0}
\end{aligned}$$

Rule B-Notification and B-OneWay describes that the unidirectional communication statements finish after having taken their send or receive transition. Rule B-SolResp and B-ReqResp describes that the bidirectional communication statements transform to runtime statements after having taken first transition in the communication. The statement solicit-response transforms to the statement wait. It finish using rule B-Wait when the reply from the called service is received. Its dual statement request-response transforms to the statement exec. In rule B-Exec the behaviour part of the request-response statement is executed.

Rule B-End-Exec describes that when the behaviour part is fully executed, the reply is sent to the caller service.

2.4.2 Service Layer

The semantic rules at the service layer are described below. The service layer can be seen as two layers since it consists of both processes and service engines. At the process level of the service layer messages are read from a message queue, while they are put into the message queue at the service level.

As we are going to see i S-Assign, S-Get and S-Start then each time a variable is altered, the nodes from the assigned data tree is copied to the variables data tree. Therefore non of the data trees in Jolie contains links.

Communication

$$(\texttt{S-Send}) \quad \frac{B \xrightarrow{\nu r\ o@l(e)} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\nu r\ o@l(e(t))} B' \cdot t \cdot \tilde{m}}$$

When an expression is sent using B-Notification or B-SolResp, the service layer takes care of evaluating the expression on the state to a data tree by rule S-Send. This is done by the function $e(t)$ in the label of the transition in the conclusion of S-Send. Recall that we write $e(t) = t'$ where e is an expression, t is a state and t' is a tree such that t' is the result of the evaluation of e in which each variable in e are looked up in the state t.

Recall that we are looking at a fragment of Jolie without communication ports. Since locations are used directly instead of being part of an communication port, we have removed the lookup of the location in the state from the rules presented in [MC11].

From S-Send we know that sending a message influences the process part of the service layer. It does not influences the service part of the service layer since it does not require spawning a process or putting a message in the message queue. Therefore S-Send-Lift is a lifting rule. We have added it to the semantics as part of this master thesis.

$$(\texttt{S-Send-Lift}) \quad \frac{P \xrightarrow{\nu r\ o@l(t)} P'}{B \triangleright_D P \xrightarrow{\nu r\ o@l(t)} B \triangleright_D P'}$$

The actions performed by a service when receiving a message are defined in the semantic rules S-Start and S-Corr. Which of the semantic rules applies for receiving a message depends on whether the received message correlates with any running process. We say that a message t' received for operation o correlates with a state t if the values of the correlation set of t' equals the values of the corresponding correlation set in t. Recall that a deployment D consists of an aliasing function α_C and an environment Γ, where α_C is used to extract information about where in a message the correlation values for an operation can be found, and Γ contains type bindings for operations known by the service. This is denoted by $D = \alpha_C \cdot \Gamma$. We write $t', \mathtt{o} \vdash_{\alpha_C} t$ when t' correlates with the process represented by t and $t, \mathtt{o} \nvdash_{\alpha_C} P$ when it does not.

$$\text{(S-Corr)}\quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad t',\mathtt{o} \vdash_{\alpha_C} t \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'}{\leq}T_i}{B \triangleright_D P \mid B'{\cdot}t{\cdot}\tilde{m} \xrightarrow{\nu r\ \mathtt{o}(t')} B \triangleright_D P \mid B'{\cdot}t{\cdot}\tilde{m}{\cdot}(r,\mathtt{o},t')}$$

$$\text{(S-Start)}\quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad t,\mathtt{o} \nvdash_{\alpha_C} P \quad B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B' \quad t'{=}\mathsf{init}(t,\mathtt{o},\alpha_C) \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'}{\leq}T_i}{B \triangleright_D P \xrightarrow{\nu r\ \mathtt{o}(t)} B \triangleright_D P \mid B'{\cdot}t_\perp {\leftarrow_{\mathtt{x}}} t {\leftarrow_{\mathtt{csets}}} t'{\cdot}\epsilon}$$

$$\mathsf{init}(t,\mathtt{o},\alpha_C) = \begin{cases} t_\perp \twoheadleftarrow_{\mathtt{p}_1} f(\mathtt{p}_1)(t) \ldots \twoheadleftarrow_{\mathtt{p}_n} f(\mathtt{p}_n)(t) & \text{if } \alpha_C(\mathtt{o}) = (\{\mathtt{p}_1,\ldots,\mathtt{p}_n\}, f) \\ t_\perp & \text{if } \mathtt{o} \notin Dom(\alpha_C) \\ \text{undefined} & \text{otherwise} \end{cases}$$

From the form of S-Corr we can see that when a received message correlates, it is added to the message queue of the correlating process. We can see from the form of S-Start that if a received message does not correlate, a new process is started. The spawned process is initialized with an empty queue, a state containing only the received message and correlation set, and the behaviour of the service after it is evaluated one step (the first step is used to synchronise with the sending of the message).

Recall that a variable is a path in the state. The creation of the state for the new process is done by updating the path `csets` in the message t with the correlation variables stored in t'. A tree with a single node with empty value denoted $t_\perp$ is then updated with a path x with the value of the result of the update of the message t. This is denoted by $t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t'$.

Our type system relies on a dynamic type check to ensure that messages received through a specific operation fulfills the requirements specified in the type declaration for the operation. Recall that we have added a dynamic type check denoted by $\vdash t' : T_{t'}$ and $T_{t'} \leq T_i$ to S-Corr and S-Start, where $\vdash t' : T_{t'}$ assigns to t' the minimal type of t'. The dynamic type check ensures that the type of a

received message is a subtype of the input type declaration of the operation it is received through. Our dynamic type checking captures the one implemented in the interpreter of Jolie [Mon10].

$$\text{(S-Get)} \quad \frac{B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B'}{B \cdot t \cdot (r,\mathtt{o},t') : x \cdot \tilde{m} \xrightarrow{\tau} B' \cdot t \leftarrow_{\mathtt{x}} t' \cdot \tilde{m}}$$

A message is read from the message queue by the process in rule S-Get. Since a variable is a path in the state, the assignment of the message t' to the variable x is done by updating path x of the state t with the message. This is denoted by $t \leftarrow_{\mathtt{x}} t'$.

$$\text{(S-Exec)} \quad \frac{B \xrightarrow{(r,\mathtt{o})!\,x} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{(r,\mathtt{o})!\,x(t)} B' \cdot t \cdot \tilde{m}}$$

The response in a bidirectional communication is sent through a shared channel. The variable which is going to be sent is evaluated on the process state to a data tree. This is described by rule S-Exec which in [MC11] is called S-SR.

$$\text{(S-Exec-Lift)} \quad \frac{P \xrightarrow{(r,\mathtt{o})!\,t} P'}{B \triangleright_D P \xrightarrow{(r,\mathtt{o})!\,t} B \triangleright_D P'}$$

$$\text{(S-Wait-Lift)} \quad \frac{P \xrightarrow{(r,\mathtt{o@}l)?t} P'}{B \triangleright_D P \xrightarrow{(r,\mathtt{o@}l)?t} B \triangleright_D P'}$$

When a connection is established through a shared channel, correlation is not used. Therefore the semantic rules for the second part of a bidirectional communication at the service engine level are lifting rules. They are added to the semantics as part of this master thesis.

$$\text{(S-Wait)} \quad \frac{B \xrightarrow{(r,\mathtt{o@}l)?x} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{(r,\mathtt{o@}l)?t'} B' \cdot t \leftarrow_{\mathtt{x}} t' \cdot \tilde{m}}$$

When a message is received as part of a solicit-response it is read directly from the shared channel which it is received over. The state is updated the same way as in S-Get. This is described by rule S-Wait which in [MC11] is called S-RR.

Other State Accesses

$$(\texttt{S-Assign}) \quad \frac{B \xrightarrow{\texttt{x} = e} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\tau} B' \cdot t \twoheadleftarrow_{\texttt{x}} e(t) \cdot \tilde{m}}$$

Except for input communication a variable can also be altered as part of an assignment. This is described in S-Assign. Recall that $e(t)$ evaluates expression e by looking up its variables in state t. The variable x is assigned the root node of the result of $e(t)$ by updating path x in the state with the root of the result of $e(t)$. This is denoted by $t \twoheadleftarrow_{\texttt{x}} e(t)$.

$$(\texttt{S-Read}) \quad \frac{B \xrightarrow{\textbf{read}\ t} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\tau} B' \cdot t \cdot \tilde{m}}$$

The rule S-Read describes the read of a process state. Since it is only a read neither the state nor the message queue is altered.

Parallel Processes and Internal Actions

$$(\texttt{S-Tau}) \quad \frac{P \xrightarrow{\tau} P'}{B \triangleright_D P \xrightarrow{\tau} B \triangleright_D P'}$$

The service engine rule S-Tau is a lifting rule. It is added as part of this master thesis.

$$(\texttt{S-Par}) \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

Parallelization of processes is described in rule S-Par. This rule is standard. It is added as part of this master thesis.

2.4.3 Network Layer

The semantic rules for networks are described below.

N-Struct

$$(\texttt{N-Struct}) \quad \frac{N_1 \equiv N_2 \quad N_2 \xrightarrow{\mu} N_2' \quad N_1' \equiv N_2'}{N_1 \xrightarrow{\mu} N_1'}$$

where structural congruence is defined as follows:

Definition 2.5 (Structural Congruence Rules at Network Layer)

$$
\begin{aligned}
(\, N_1 \mid N_2 \,) \mid N_3 &\equiv N_1 \mid (\, N_2 \mid N_3 \,) \\
N_1 \mid N_2 &\equiv N_2 \mid N_1 \\
N \mid 0 &\equiv N \\
\text{if } r \notin fn(N_2): \quad ((\boldsymbol{\nu} r\,) N_1) \mid N_2 &\equiv (\boldsymbol{\nu} r\,)(N_1 \mid N_2)
\end{aligned}
$$

N-Comm

$$
\text{(N-Comm)} \quad \frac{S_1 \xrightarrow{\boldsymbol{\nu} r\, \circ @ l_2(t)} S_1' \quad S_2 \xrightarrow{\boldsymbol{\nu} r\, \circ(t)} S_2' \quad r \notin \mathtt{cn}(S_1) \cup \mathtt{cn}(S_2)}{[S_1]_{l_1} \mid [S_2]_{l_2} \xrightarrow{\tau} \boldsymbol{\nu} r\, (\, [S_1']_{l_1} \mid [S_2']_{l_2} \,)}
$$

The rule N-Comm describes a unidirectional communication or the first part of a bidirectional communication. Its premises make sure that the service at the receiver location receives the message and that the setup of a shared channel is done with a channel which name is not already used in any of the concerned services.

N-Response

$$
\text{(N-Response)} \quad \frac{S_1 \xrightarrow{(r, \circ @ l_1)?t} S_1' \quad S_2 \xrightarrow{(r, \circ)!\, t} S_2'}{\boldsymbol{\nu} r\, (\, [S_1]_{l_1} \mid [S_2]_{l_2} \,) \xrightarrow{\tau} [S_1']_{l_1} \mid [S_2']_{l_2}}
$$

The rule N-Response describes the second part of a bidirectional communication which is done through the shared channel setup in the conclusion of N-Comm.

In both N-Comm and N-Response we see that the communication between two services is seen as an internal action for the rest of the network.

N-Tau and N-Par

$$
\text{(N-Tau)} \quad \frac{S \xrightarrow{\tau} S'}{[S]_l \mid N \xrightarrow{\tau} [S']_l \mid N}
$$

$$
\text{(N-Par)} \quad \frac{N_1 \xrightarrow{\mu} N_1'}{N_1 \mid N_2 \xrightarrow{\mu} N_1' \mid N_2}
$$

The rules N-Tau and N-Par are standard and describe that the change of a part of a network applies to the whole network.

N-Restriction

$$(\texttt{N-Restriction}) \quad \frac{N \xrightarrow{\tau} N'}{\nu r\,(\ N\ ) \xrightarrow{\tau} \nu r\,(\ N'\ )}$$

The rule N-Restriction is standard and describes that the change of a network also applies to the restricted network. The rule is added as part of this master thesis.

Chapter 3

Type System for Jolie

This chapter presents the static type system. The typing environment is presented in 3.1, the typing rules are presented in 3.2 and the properties type preservation and type safety are presented in respectively 3.3 and 3.4. Notice, that the subtyping rules are presented in 2.3 since they are used by the dynamic type check. All the typing rules are reported also in a single table in Appendix A, for reference.

The type system follows a simple principle: Recall that variables in Jolie are not declared. Alternation of a type tree by extending it is therefore not considered a type error. Thereby is alternation of a node in a type tree considered a type error.

3.1 Typing Environment

Recall that a typing environment, Γ is a set consisting of type bindings for variables and operations. The typing environment has different purposes at the different layers. At the behavioural layer it is used for operation type bindings declared in the deployment part of the process and for variable type bindings inferred in the behavioural part of the process. At the service layer it is used

for operation type bindings declared for the service, while at the network layer it is used for operation type bindings declared for the network.

A type environment is at the following form:

$$
\begin{aligned}
\Gamma ::= \quad & \texttt{o@}l : \texttt{<}T\texttt{>} \\
| \quad & \texttt{o@}l : \texttt{<}T, T\texttt{>} \\
| \quad & \texttt{o} : \texttt{<}T\texttt{>} \\
| \quad & \texttt{o} : \texttt{<}T, T\texttt{>} \\
| \quad & x : T \\
| \quad & \emptyset \\
| \quad & \Gamma, \Gamma
\end{aligned}
$$

We remind that a type declaration for an operation has the form ***key*** : <*type*>. For input communication the key is the operation name, while it for output communication consists of both the operation name and a location of the hosting service. The type is a single type tree for unidirectional communication and a pair of type trees for bidirectional communication.

The type declaration for a variable has the form $x : T$ where x is the root of a variable path and T is the type of x. We assume that operations and variables are not allowed to have the same name. This assumption is made solemnly because the typing environment is a set, and therefore the key must be unique. The assumption can for instance be realized by adding a character which a developer can not type to either each variable name or to each operation name. It is omitted from this thesis for clarity.

We use the standard set operations $key \in \Gamma$ and $key : type \in \Gamma$ for detecting whether respectively a key and a binding is member of an environment. Since the key of a type declaration for a variable is the root of the variable, and the subsequent part of the variable and its type binding is part of the type tree for the root, we use the following shortcuts for looking up variable type bindings in an environment Γ:

$$
x \in \Gamma = \begin{cases} \text{true} & \text{if } r(x) : T \text{ is a type binding in } \Gamma \ \wedge \ x \in T \\ \text{false} & \text{otherwise} \end{cases} \tag{3.1}
$$

$$
x : T \in \Gamma = \begin{cases} \text{true} & \text{if } r(x) : T' \text{ is a type binding in } \Gamma \ \wedge \ x : T \in T' \\ \text{false} & \text{otherwise} \end{cases} \tag{3.2}
$$

in which we write $r(x)$ for the root edge of path x:

Definition 3.1 (Root of a Path) $r(x) = a$ iff $a.x' = x$.

where x and x' are paths and a is an edge.

The functions $x \in \Gamma$ and $x : T \in \Gamma$ make use of the function $x : T \in T'$. Let T and T' be two types and x and x' be two variables, where x' is the part of x excluding its root. We write $x : T \in T'$ when x' is a path in T' with the subtree T:

$$x : T \in T' = \begin{cases} \text{true} & \text{if } r(x) = x \\ \text{true} & \text{if } r(x) \neq x \ \wedge \ x = r(x).x' \ \wedge \ x' \text{ is a path in } T' \\ & \text{pointing to the subtree } T \\ \text{false} & \text{otherwise} \end{cases} \tag{3.3}$$

Note that if x only consists of one edge $x : T \in T' = \text{true}$ since the root of a variable is not part of its type.

Similarly we write $x \in T$ when x' is a path in T:

$$x \in T = \begin{cases} \text{true} & \text{if } r(x) = x \\ \text{true} & \text{if } r(x) \neq x \ \wedge \ x = r(x).x' \ \wedge \ x' \text{ is a path in } T \\ \text{false} & \text{otherwise} \end{cases} \tag{3.4}$$

For the dual function of $x \in \Gamma$ we write $x \notin \Gamma$ when there exists no type bindings for x in Γ:

Definition 3.2 $x \notin \Gamma = \nexists T.\ x : T \in \Gamma$

Furthermore are $o \notin \Gamma$, $o@l \notin \Gamma$ and $x \notin T$ defined similar.

3.2 Typing Rules

The structure of the type system follows the layers of Jolie. The typing rules are therefore presented in the layers.

3.2.1 Type Checking of the Behavioural Layer

The statements at the behavioural layer are called behaviours. If a behaviour B is typed with respect to an environment Γ and updates Γ to Γ' during type checking we write $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$. The judgement has form of a Hoare triple such that the update of an environment can be distributed to other parts of the type checking tree, than a possible subtree of the conclusion where it is added. A difference from the normal use of a Hoare triple is that the invariant is at the right side instead of the left side.

The following rules are used for type checking behaviours:

T-Nil The typing rule for a nil behaviour is an axiom. In the conclusion the typing environment is not changed, since the nil statement doesn't affect the typing environment.

$$(\texttt{T-Nil}) \quad \frac{}{\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma}$$

T-If-Then-Else The rule for typing an if statement is standard: An if statement is typable if its condition has type bool, and if the type checking of its branches perform the same updates to the environment. We require the branches to perform the same updates because we do not know which branch will be taken.

$$(\texttt{T-If-Then-Else}) \quad \frac{\Gamma \vdash e{:}\mathit{bool} \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \texttt{if}(e)\ B_1\ \texttt{else}\ B_2 \triangleright \Gamma'}$$

In Jolie the else part is optional. To avoid writing too many similar typing rules we have omitted it here, because it is syntactic sugar for the case $\texttt{if}(e)\ B_1\ \texttt{else}\ \mathbf{0}$.

The premise $\Gamma \vdash e : \mathit{bool}$ requires that expression e is type checked against type textttbool. In general, we write $\Gamma \vdash e : T$ when the result of the evaluation of expression e under type environment Γ has minimal type T.

T-While The rule for typing a while statement is standard: A while statement is typable if its condition has type bool, and if type checking its body has no influence on the typing environment.

$$\text{(T-While)} \quad \frac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\ \{B\} \triangleright \Gamma}$$

Above, we require that the body of the while loop does not change the typing of variables because we do not know whether the body will be executed at all, and for how many times. We also require that expression e is type checked against type textttbool.

T-Choice The rule for typing a choice statement is standard: A choice statement is typable if the type checking of all its branches perform the same updates to the environment.

$$\text{(T-Choice)} \quad \frac{\forall j \in J \ .\ \Gamma \vdash_{\mathrm{B}} \eta_j ; B_j \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \Gamma'}$$

Above, we require that the choice options perform the same updates to the environment since we do not know at compile time which option is chosen.

In the premise the guard and the body is rewritten as sequential because, both are being executed in sequence according B-Choice.

T-Par A parallel behaviour is typable if each of its threads are well typed and if the updated environments from type checking its threads with respect to two disjoint environments are disjoint.

$$\text{(T-Par)} \quad \frac{\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2' \quad \mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'}$$

We write respectively Γ, Γ' and $\Gamma \uplus \Gamma'$ for the disjoint union of two environments Γ and Γ'. In the conclusion of T-Par we require the respectively Γ_1 and Γ_2 and Γ_1' and Γ_2' to be disjoint in order to avoid dependencies in B_1 and B_2. The disjunction goes for whole variable trees insted of just for paths as expressed in the premise $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$. We write $\mathit{Roots}(\Gamma)$ for the set containing the root edge of each variable in an environment Γ:

$$\mathit{Roots}(\Gamma) = \{r(x) | x : T \in \Gamma\}$$

Notice, that $\mathit{Roots}(\Gamma)$ does not contain operations, since an operation is not a path. The premise $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$ sets the requirement that this

rule can only be applied to behaviours which environments are splitted such that there exists no overlap in the splitted environments, neither in the two environment parts that the behaviour is typed with respect to (Γ_1 and Γ_2), nor in the two resulting environment parts (Γ'_1 and Γ'_2). The disjunctions goes for whole variables and not only parts of variables to avoid situations where one thread relies on a variable to have a given structure, while the other thread alters the structure of the variable. For instance if `o@l(e)` $\mid x.b = 4$ where `o@l` : <T> and $T : int\{.b[0,1] : string\}$.

The requirement that the environments which the behaviours are type checked with respect to, must be disjoint are implicit in $Roots(\Gamma'_1) \cap Roots(\Gamma'_2) = \emptyset$ since a type binding can never be removed from the environment according to the form of the typing system and the transition function *sideEffect* which will be defined in 3.3.3 (definition 3.12).

T-Seq A sequence statement typed with respect to an environment is typable if its first component is typable with respect to the environment and its second component is typable with respect of the update of the environment performed by the first component. The update of the environment performed by the sequence statement is the update performed by the second component with respect to the update performed by the first component.

$$(\texttt{T-Seq}) \quad \frac{\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma' \quad \Gamma' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma''}{\Gamma \vdash_{\mathrm{B}} B_1 ; B_2 \triangleright \Gamma''}$$

T-Notification A notification statement is typable if the environment contains type information for the operation and if its expression is typable and has a type which is a subtype of the type of the operation. Since sending an expression doesn't provide any changes to the type environment, the type environment it not updated.

$$(\texttt{T-Notification}) \quad \frac{\texttt{o@}l{:}\texttt{<}T_o\texttt{>} \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)} \triangleright \Gamma}$$

The typing rule for notification ensures that what is send shall always be defined. The small letter used in combination with T is solemnly to help the reader remember what the different type trees are bound to. It does not specify the form of the type tree, nor whether two type trees are equal.

3.2.1.1 Alternation of the typing environment

The assign statement, the input and output statements and the wait statement update the typing environment. The typing rules for each of these statements follows the same structure: Assignment of variables which has no type binding in the typing environment is typed by a rule with "New" in its name, while assignment of variables which types are not changed is typed by a rule with "Exists" in its name. That way the type system ensures that a statement in which the basic type of a variable is altered, is not typable.

The updates of the typing environment are performed by the function *upd* which makes use of the function *addPath* to add a path to a variable in the typing environment. The function *addPath* makes use of the function *addChild* to extend a path with a child. We follow the chain of dependencies and describe the three functions starting with *addChild*:

Recall that $r(x)$ denotes the root edge of path x. The function *addChild* takes a type T and a type binding $x : T_x$ consisting of a path x and a type T_x. It adds path x with type T_x to path $p(x)$ in T.

Definition 3.3 (*addChild*)

$$addChild(T, x : T_x) = T' \text{ where } \forall x' \in T \,.\, x' \in T' \text{ and } x : T_x \in T'$$

The function is undefined if

$$x : T'_x \in T \text{ where } T'_x \neq T_x$$

and if

$$p(x) \notin T \ \wedge \ p(x) \neq r(x)$$

where we let $p(x)$ denote the path to the parent of the node pointed to by x:

Definition 3.4 (Parent of a Path) We write $p(x)$ for a function which takes a path x and returns the path without its leaf. If x is root $\perp$ is returned.

When a variable is given a value, where one or more predecessors of the variable's path haven't been given any value, the predecessors get the type `void`. The function *addPath* takes a path x and two types T and T_x. It steps through x from its leaf a and builds up the type tree for x, where the leaf of x gets type T_x and the missing predecessors get type `void`. When it reaches an existing predecessor of x it connects the builded tree to the predecessor using *addChild*.

Definition 3.5 (*addPath*)

$$addPath(T, x, T_x) = \begin{cases} addPath(T, p(x) : \texttt{void}\{.a : T_x\}) & \text{if } p(x) \notin T \wedge x = p(x).a \\ addChild(T, x : T_x) & \text{if } p(x) \in T \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.5}$$

The function *upd* takes as input an typing environment Γ, a variable x and a type T_x. It updates the part x of Γ with type T_x. The form of the output differs regarding the form of Γ: If the root of variable x is in Γ, then the root is bound to a type tree representing x with type T_x and in which all missing predecessors of x are assigned type `void`. This type tree is built by *addPath*. If the root of x is not in Γ and if x is not the root of itself, a similar type tree is build up by *addPath* but instead of building of an existing type tree, a new consisting of a single node with type `void` is used. If x is an edge to a root node, and if it is not in Γ, the type T_x is assigned directly to x without use of help functions.

Definition 3.6 (*upd*)

$$upd(\Gamma, x, T_x) = \begin{cases} \Gamma[r(x) \mapsto addPath(T_{r(x)}, x, T_x)] & \text{if } r(x) : T_{r(x)} \in \Gamma \\ \Gamma[r(x) \mapsto addPath(\texttt{void}, x, T_x)] & \text{if } r(x) \notin \Gamma \wedge r(x) \neq x \\ \Gamma[x \mapsto T_x] & \text{if } r(x) \notin \Gamma \wedge r(x) = x \end{cases} \tag{3.6}$$

T-Assign-New An assign statement for which the environment doesn't contain type information for the variable, is typable if the expression can be type checked against a type under the environment. The corresponding update of the environment is performed by *upd*.

$$(\texttt{T-Assign-New}) \quad \frac{\Gamma \vdash e : T_e \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{x = e} \triangleright upd(\Gamma, x, bt(T_e))}$$

Recall from the semantics that S-Assign only assign the value of the root node of a data tree to a variable. We call the value of a node in a type tree for the basic type of the node. The basic type of a type tree is the basic type of its root node. In T-Assign-New the typing environment is updated with the basic type of the type of the expression. This is denoted by $bt(T_e)$.

T-Assign-Exists An assign statement for which the environment contains type information for the variable, is typable if the expression with respect to

the environment can be type checked against a type which basic type equals the basic type of the type tree bound to the variable in the environment.

$$(\texttt{T-Assign-Exists}) \quad \frac{\Gamma \vdash e{:}T_e \quad x{:}T_x \in \Gamma \quad bt(T_e){=}bt(T_x)}{\Gamma \vdash_{\mathrm{B}} \texttt{x = e} \rhd \Gamma}$$

Only the basic type is considered since only the root node of the data tree is assigned in an assign statement.

T-OneWay-New An one way statement for which the environment does not contain type information for the variable, is typable if the environment contains type information for the operation. The corresponding update of the environment is performed by *upd*. The variable is assigned the type of the operation.

$$(\texttt{T-OneWay-New}) \quad \frac{\texttt{o}{:}\langle T_i \rangle \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \rhd upd(\Gamma, x, T_i)}$$

T-OneWay-Exists An one-way statement for which the environment contains type information for the variable, is typable if the environment contains type information for the operation and if the operation type is a subtype of the variable type. Since the variable already has an declaration in the type environment, there is no need to update the environment with the same declaration.

$$(\texttt{T-OneWay-Exists}) \quad \frac{\texttt{o}{:}\langle T_i \rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \rhd \Gamma}$$

Reading whole data trees into variables raises the possibility to read a data tree which type has paths which extend the variable type and paths which don't. This behaviour is not wished in Jolie and the type system avoids it by only allowing a data tree to be read into a variable tree if the data tree is a subtype of the variable tree.

T-SolResp-New and T-SolResp-Exists Since the evaluation of a solicit-response statement first send a message and thereafter receives a message, the typing rules for a solicit-response statement have the same premises as the rules for notification and one-way, and for the same reasons.

$$(\texttt{T-SolResp-New}) \quad \frac{\texttt{o@}l{:}\langle T_o, T_i \rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \rhd upd(\Gamma, x, T_i)}$$

$$\text{(T-SolResp-Exists)} \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma}$$

T-ReqResp-New and T-ReqResp-Exists Since the behaviour B of a statement `o(x)(x')`$\{B\}$ is executed between two communications, it must be typable with respect to Γ updated with the change performed by the first communication. Furthermore the type binding for the variable to be send in the second communication is looked up in the resulting environment of the type checking of B. The rest of the request-response rules are similar to the typing rules for one-way and notification and for the same reasons.

$$\text{(T-ReqResp-New)} \quad \frac{\texttt{o}{:}\langle T_i, T_o\rangle \in \Gamma \quad x \notin \Gamma \quad upd(\Gamma, x, T_i) \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x'{:}T_{x'} \in \Gamma' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\ \{B\} \triangleright \Gamma'}$$

$$\text{(T-ReqResp-Exists)} \quad \frac{\texttt{o}{:}\langle T_i, T_o\rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x'{:}T_{x'} \in \Gamma' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\ \{B\} \triangleright \Gamma'}$$

3.2.1.2 Type Checking of Run-Time Statements

Each of the bidirectional communication statements evaluates to a run-time statement when performing the first communication. As a consequence the type checking of the corresponding run-time statement is similar to the type checking of the part of the communication statement which excludes the first communication.

T-Wait-New and T-Wait-Exists Recall from the semantics that a request-response statement after a step of evaluation becomes a wait statement. Since the step of evaluation handles the sending of a message, the type checking of a wait statement is similar to the type checking of the second communication performed by a solicit-response.

$$\text{(T-Wait-New)} \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright upd(\Gamma, x, T_i)}$$

$$\text{(T-Wait-Exists)} \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \Gamma}$$

T-Exec Recall from the semantics that a request-response statement after a step of evaluation becomes an exec statement. Since the step of evaluation handles the receiving of an incoming message, the type checking of an exec

statement i similar to the part of type checking a request-response statement which type checks the behaviour and the variable which is going to be send as response. Hence, the rules are similar.

$$(\texttt{T-Exec}) \quad \frac{\texttt{o}{:}\langle T_i, T_o\rangle \in \Gamma \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x{:}T_x \in \Gamma' \quad T_x \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r,\texttt{o},\texttt{x},B) \triangleright \Gamma'}$$

3.2.2 Type Checking of the Service Layer

We will for a moment break the topdown approach and present typing of services before presenting typing of processes. This is done since typing of services reveal some knowledge of the content of the typing environment which we use in the typing rules for processes.

3.2.2.1 Type Checking of Services

Recall that a service S has the form:

$$S ::= \quad B \triangleright_D P$$

for a behaviour B, a deployment D and a process P. We write$\Gamma \vdash_{\mathrm{S}} S$ if a service S is typed with respect to a typing environment Γ. The following rule is used for type checking services:

T-Service A service is typable with respect to an environment if its behaviour and process is typable with respect to the environment and if the environment consists of the operation type bindings known by the service.

$$(\texttt{T-Service}) \quad \frac{D = \alpha_C \cdot \Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P}$$

Recall that a deployment D consists of an aliasing function α_C and an environment Γ, where α_C is used to extract information about where in a message the correlation values for an operation can be found, and Γ contains type bindings for operations known by the service. Premise $D = \alpha_C \cdot \Gamma$ makes the restriction that the behaviour and the process of a service is typed with respect to the operation type declarations known by the service. The typejudgements used in premise $\Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma'$ and $\Gamma \vdash_{\mathrm{P}} P$ are described in the two following sections.

Since the behaviour is replicated when spawning processes it is not typed with respect to an environment which contains variable type bindings.

3.2.2.2 Type Checking of Service Layer Behaviours

The behaviour B of a service $B \triangleright_D P$ can be either a choice or a **0** according to the grammar defined in [MC11]. It is therefore not a problem not knowing which branch of a choice is chosen, and thereby how the environment looks since the environment is not further used. Therefore we can let the type system accept a brighter amount of programs and this is the reason we have made a new typing relation instead of reusing $\vdash_B$. The form of a judgement for typing behaviours at the service layer is $\Gamma \vdash_{\text{BSL}} B \triangleright \Gamma'$ for a behaviour B and two environments Γ and Γ'.

As discussed above the following two rules are the only ones needed for typing behaviours at the service layer:

T-BSL-Nil The typing rule for a nil behaviour is an axiom.

$$(\texttt{T-BSL-Nil}) \quad \frac{}{\Gamma \vdash_{\text{BSL}} \mathbf{0} \triangleright \Gamma}$$

T-BSL-Choice A choice behaviour is typable with respect to a typing environment Γ if the resulting typing environments from type checking the branches do not contain two different type bindings of the same variable.

$$(\texttt{T-BSL-Choice}) \quad \frac{\forall j \in J \;.\; \Gamma \vdash_{\text{B}} \eta_j ; B_j \triangleright \Gamma_j \qquad \nexists \mathrm{T}_k, \mathrm{T}_l \;.\; x{:}\mathrm{T}_k \in \bigcup \Gamma_j \wedge x{:}\mathrm{T}_l \in \bigcup \Gamma_j \wedge \mathrm{T}_k \neq \mathrm{T}_l}{\Gamma \vdash_{\text{BSL}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \bigcup \Gamma_{j \in J}}$$

As discussed above a choice behaviour at the service layer can not be part of a sequence or a parallel behaviour and thereby is the resulting environment of evaluating a choice behaviour not further used.

3.2.2.3 Type Checking of Processes

Recall that a process either consists of other processes in parallel or of a behaviour, a state and a message queue or is the nil process:

$$P ::= \quad B \cdot t \cdot \tilde{m} \quad | \quad \mathbf{0} \quad | \quad P \mid P$$

We write $\Gamma \vdash_{\mathrm{P}} P$ if a process P is typed with respect to an environment Γ. From the typing rules for services we know that Γ only contains type bindings for operations.

The following rules are used for type checking processes:

T-Process-Nil The typing rule for a nil process is an axiom.

$$(\texttt{T-Process-Nil}) \quad \frac{}{\Gamma \vdash_{\mathrm{P}} \mathbf{0}}$$

T-Process-Par A parallel process is typable with respect to an environment if each of the processes it consists of is typable with respect to the environment.

$$(\texttt{T-Process-Par}) \quad \frac{\Gamma \vdash_{\mathrm{P}} P_1 \quad \Gamma \vdash_{\mathrm{P}} P_2}{\Gamma \vdash_{\mathrm{P}} P_1 \mid P_2}$$

The service has knowledge of operation type bindings and therefore is that knowledge accessible for all of the service's processes. Since the knowledge of variable type bindings is internal in a process T-Process-Par does not require disjoint union of the environment which P_1 and P_2 is typed with respect to.

T-Process A process consisting of a behaviour B, a state t and a message queue $\tilde{m}$ is typable with respect to a typing environment Γ if each of its components are typable with respect to the same environment union an environment Γ' which does not contains type bindings for operations.

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma' \vdash_{\mathrm{BSL}} B \triangleright \Gamma'' \quad \Gamma,\Gamma' \vdash_{\mathrm{state}} t \quad \Gamma,\Gamma' \vdash_{\mathrm{queue}} \tilde{m} \quad \nexists \mathtt{o}.\, \mathtt{o@}l{:}\langle O\rangle \in \Gamma' \vee \mathtt{o}{:}\langle O\rangle \in \Gamma'}{\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot \tilde{m}}$$

The restriction on Γ' is set by premise $\nexists \mathtt{o}.\, \mathtt{o@}l : \langle O\rangle \in \Gamma' \vee \mathtt{o} : \langle O\rangle \in \Gamma'$ since we let O denote a type which can either be a type tree or a pair of type trees. The form of Γ' ensures that the components of a process are not typed under a broader knowledge of operation type declarations than the service is. It also

ensures that Γ and Γ' are disjoint since we know from the type checking rule for services that Γ only contains operation type bindings.

Premise $\Gamma, \Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma''$ forces Γ' to contain type information for variables known by the process. This will be clear when we present type preservation for behaviours in 3.3.4.

A state t' is typed with respect to a typing environment $\Gamma_{t'}$ when the type for each of the root variables in $\Gamma_{t'}$ is a supertype to a variable with the same name in t' (1) and when each of the root variables in t' has a type binding in $\Gamma_{t'}$ (2).

Definition 3.7 (Type Checking of a State) We write $\Gamma_{t'} \vdash_{\text{state}} t'$ if and only if

1. $\forall a : T_a \in \mathit{Roots}(\Gamma_{t'}). \vdash a(t') : T \wedge T \leq T_a$
2. $\forall a \in \mathit{Roots}(t'). a \in \mathit{Roots}(\Gamma_{t'})$

where a ranges over edges, T and T_a are types, $\Gamma_{t'}$ is an environment and t' is a process state.

Note that the typing information for operations are excluded from the definition since an operation name is not a path.

Recall that we write $\mathit{Roots}(\Gamma_r)$ for the set containing the root edge of each variable in an environment Γ_r:

$$\mathit{Roots}(\Gamma_r) = \{r(x) | x : T \in \Gamma_r\}$$

Notice, that $\mathit{Roots}(\Gamma'_t)$ does not contain operations, since an operation is not a path. By the form of the typing rules for behaviours which alternates the typing environment and by the form of $\mathit{Roots}(\Gamma_r)$ we know that in 1 a is the first edge in a variable name, and T_a is the type tree for a. Recall that the minimal type T of tree t'' can be assigned to t'' with $\vdash t'' : T$. In 1 each variable in a state is compared by its whole structure with the type binding from the environment.

We write $\mathit{Roots}(t_r)$ for the set containing the root edge of each variable in a state t:

$$\mathit{Roots}(t_r) = \{r(x') | x \text{ is a path in } t_r \ \wedge \ x = r(x).x'\}$$

where x and x' are paths and t_r is a state. Since the state consists of a empty node which childrens are variables in the state, the root of a path x in t_r is cut off in order to find the root of the variable of path x. By the form of $\mathit{Roots}(t_r)$ and $\mathit{Roots}(\Gamma_r)$ we know that by 2 a state is not typable with respect to a typing environment if it contains one or more variables which first edge does not have any type binding in the environment.

A message queue $\tilde{m}$ is typed with respect to a typing environment Γ_m when for all messages in $\tilde{m}$, the type for the data tree in the message is a subtype of the required type for the receive operation, with respect to typing environment Γ_m.

Definition 3.8 (Type Checking of a Message Queue) We write $\Gamma_m \vdash_{\text{queue}} (r_j, o_j, t_j)_{j \in J}$ if and only if

$$\begin{aligned} \forall j \in J.(\texttt{<o}_j\texttt{>} : T_j \in \Gamma_m \ \vee \ \texttt{<o}_j\texttt{>} : \texttt{<}T_j, T'_j\texttt{>} \in \Gamma_m) \wedge \\ \vdash t_j : T''_j \ \wedge \ T''_j \leq T_j \end{aligned}$$

where Γ_m is a typing environment and T ranges over types, t ranges over data trees, o ranges over operations and r ranges over channels.

3.2.3 Type Checking of the Network Layer

The statements at the network layer are called networks. We write $\Gamma \vdash_{\text{N}} N$ if a network N is typable with respecto to a typing environment Γ. The following rules are used for type checking networks:

T-Network-Nil The typing rule for a nil process is an axiom.

$$(\texttt{T-Network-Nil}) \quad \frac{}{\Gamma \vdash_{\text{N}} \mathbf{0}}$$

T-Deployment A service deployet on a location is typable with respect to an environment if it is typable with respect to a related environment which does not contain any output operations to the service.

$$(\texttt{T-Deployment}) \quad \frac{\Gamma \vdash_{\text{S}} B \triangleright_D P \quad l \notin \mathit{locs}(\Gamma)}{\{\texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma\} \cup \{\texttt{o@}l : \texttt{<}O\texttt{>} \mid \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma\} \vdash_{\text{N}} [B \triangleright_D P]_l}$$

Recall from the typing rules for the network layer that the typing environment which a service is typed with respect to consists of type declarations for operations known by the service. Premise $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ restricts Γ to only consist of operation type bindings.

We write $locs(\Gamma_l)$ for the set of locations which appear in operation bindings in an environment Γ_l:

$$locs(\Gamma) = \{l | \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma\}$$

Premise $l \notin locs(\Gamma)$ restricts Γ to not including self calls, since self calls are not allowed according to the semantics.

The environment in the conclusion equals the environment in the premises except that the keys for the output operation type bindings are converted to same format as the input operations. We make use of this convertion in T-Network.

T-Network Two networks running in parallel are typable with respect to an environment if each of them is typable with respect to a subset of the environment and if

$$(\texttt{T-Network}) \quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} N_1 \quad \Gamma_2 \vdash_{\mathrm{N}} N_2 \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2}$$

We write $locs(N)$ for the set of locations which services in a network N are deployed at. We let $locs(N)$ be recursively defined:

$$\begin{aligned} locs([S]_l) &= \{l\} \\ locs(\boldsymbol{\nu} r\ N) &= locs(N) \\ locs(N_1 \mid N_2) &= locs(N_1) \cup locs(N_2) \\ locs(\mathbf{0}) &= \emptyset \end{aligned}$$

Premise $\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1$ where $l \in locs(N_2).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1)$ requires that all calls to a service in the other network has an matching type binding in the environment which the other network is typed under. It also requires that a location is unique. Premise $\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2$ where $l \in locs(N_1).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2)$ is similar.

Since service-oriented architectures are considered open architectures a provider of an operation might not be a part of the network which is type checked. It

is therefore necessary also to require whether there exists no type mismatches between any of the operation declaration in the environments under which the two parallel networks are typed, in order to union the environments. This is done in premise $\neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)$

T-Restriction A restricted network is typable with respect to an environment if the network is typable with respect to the environment, since channel restriction has no influence on the type checking.

$$(\texttt{T-Restriction}) \quad \frac{\Gamma \vdash_{\mathrm{N}} N}{\Gamma \vdash_{\mathrm{N}} \boldsymbol{\nu} r\, N}$$

3.3 Type Preservation

The property that if a well typed statement takes a transition then the resulting statement is also well typed, is called type preservation. Recall that the type judgement for behaviours is a Hoare triple consisting of a initial environment, a behaviour and a resulting environment, where the resulting environment is the initial environment updated with the changes performed during the type checking of the behaviour. The type preservation property therefore requires for behaviours that the resulting environment for a behaviour after it takes a transition is equal to the resulting environment for the behaviour before it takes the transition.

We present type preservation theorems for each of the layers in Jolie in 3.3.4. Before we present the preservation theorems we present a transition function which is used in the type preservation theorems. The transition function updates the typing environment with respect to the action of the transition. In order to address typability before and after a transition it is necessary to specify a transition function because the semantic of Jolie is described in a labeled transition system. The transition function is presented in 3.3.3.

Before presenting the preservation theorems we also present the Inversion Lemma and Structural Congruence Lemmas, which are used in the proofs for the type preservation theorems. the Inversion Lemma shows that if we have a statement typed with respect to an environment, then we must also have the premises necessary to type the statement. It is presented in 3.3.1. The Structural Congruence lemmas show that two structural congruent statements are typable with respect to the same environment. For behaviours we furthermore require that

they also make the same updates to the environment during type checking. The Structural Congruence Lemmas are presented in 3.3.2.

3.3.1 Inversion of the Typing Relation

We know from the form of the type system that it is unambiguous. We therefore know that if we have a statement typed with respect to an environment then we must also have the premises necessary to type the statement:

Lemma 3.9 (Inversion of the Typing Relation)

1. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{x=}e \triangleright \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))$ *then* $\Gamma \vdash e : T_e$ *and* $x \notin \Gamma$.
2. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{x=}e \triangleright \Gamma$ *then* $\Gamma \vdash e : T_e$, $x : T_x \in \Gamma$ *and* $\mathit{bt}(T_e) = \mathit{bt}(T_x)$.
3. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{if(}e\texttt{)}\ B_1\ \texttt{else}\ B_2 \triangleright \Gamma'$ *then* $\Gamma \vdash e : \mathit{bool}$, $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'$ *and* $\Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'$.
4. *If* $\Gamma \vdash_{\mathrm{B}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \Gamma'$ *then* $\forall j \in J\ .\ \Gamma \vdash_{\mathrm{B}} \eta_j; B_j \triangleright \Gamma'$.
5. *If* $\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'$ *then* $\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1'$, $\Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2'$ *and* $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$.
6. *If* $\Gamma \vdash_{\mathrm{B}} B_1; B_2 \triangleright \Gamma''$ *then* $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'$ *and* $\Gamma' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma''$.
7. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{while(}e\texttt{)}\{B\} \triangleright \Gamma$ *then* $\Gamma \vdash e : \mathit{bool}$ *and* $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma$.
8. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o@l(}e\texttt{)} \triangleright \Gamma$ *then* $\texttt{o@l} : \texttt{<}T_o\texttt{>} \in \Gamma, \Gamma \vdash e : T_e$ *and* $T_e \leq T_o$.
9. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o@l(}e\texttt{)(x)} \triangleright \mathit{upd}(\Gamma, x, T_i)$ *then* $\texttt{o@l} : \texttt{<}T_o, T_i\texttt{>} \in \Gamma, \Gamma \vdash e : T_e$, $T_e \leq T_o$ *and* $x \notin \Gamma$.
10. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o@l(}e\texttt{)(x)} \triangleright \Gamma$ *then* $\texttt{o@l} : \texttt{<}T_o, T_i\texttt{>} \in \Gamma, \Gamma \vdash e : T_e$, $T_e \leq T_o$, $x : T_x \in \Gamma$ *and* $T_i \leq T_x$.
11. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \mathit{upd}(\Gamma, x, T_i)$ *then* $\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma$ *and* $x \notin \Gamma$.
12. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \Gamma$ *then* $\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma, x : T_x \in \Gamma$ *and* $T_i \leq T_x$.
13. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')\{}B\texttt{\}} \triangleright \Gamma'$ *then* $\texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma$, $x \notin \Gamma$, $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} B \triangleright \Gamma'$, $x' : T_{x'} \in \Gamma'$ *and* $T_{x'} \leq T_o$.
14. *If* $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')\{}B\texttt{\}} \triangleright \Gamma'$ *then* $\texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma, x : T_x \in \Gamma$, $T_i \leq T_x$, $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$, $x' : T_{x'} \in \Gamma'$ *and* $T_{x'} \leq T_o$.

15. *If* $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, o@l, x) \triangleright \mathit{upd}(\Gamma, x, T_i)$ *then* $o@l : \langle T_o, T_i \rangle \in \Gamma$ *and* $x \notin \Gamma$.

16. *If* $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, o@l, x) \triangleright \Gamma$ *then* $o@l : \langle T_o, T_i \rangle \in \Gamma$, $x : T_x \in \Gamma$ *and* $T_i \leq T_x$.

17. *If* $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, o, x, B) \triangleright \Gamma'$ *then* $o : \langle T_i, T_o \rangle \in \Gamma$, $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$, $x : T_x \in \Gamma'$ *and* $T_x \leq T_o$.

18. *If* $\Gamma \vdash_{\mathrm{BSL}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \bigcup \Gamma_{j \in J}$ *then* $\forall j \in J \,.\, \Gamma \vdash_{\mathrm{B}} \eta_j; B_j \triangleright \Gamma_j$ *and* $\nexists T_k,\ T_l \,.\, x : T_k \in \bigcup \Gamma_j \wedge x : T_l \in \bigcup \Gamma_j \wedge T_k \neq T_l$.

19. *If* $\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot \tilde{m}$ *then* $\Gamma, \Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma''$, $\Gamma, \Gamma' \vdash_{\mathrm{state}} t$, $\Gamma, \Gamma' \vdash_{\mathrm{queue}} \tilde{m}$ *and* $\nexists o.\ o@l : \langle O \rangle \in \Gamma' \vee o : \langle O \rangle \in \Gamma'$.

20. *If* $\Gamma \vdash_{\mathrm{P}} P_1 \mid P_2$ *then* $\Gamma \vdash_{\mathrm{P}} P$ *and* $\Gamma \vdash_{\mathrm{P}} P$.

21. *If* $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ *then* $D = \alpha_C \cdot \Gamma$, $\Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma'$ *and* $\Gamma \vdash_{\mathrm{P}} P$.

22. *If* $\{o@l' : \langle O \rangle \in \Gamma\} \cup \{o@l : \langle O \rangle | o : \langle O \rangle \in \Gamma\} \vdash_{\mathrm{N}} [B \triangleright_D P]_l$ *then* $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ *and* $l \notin \mathit{locs}(\Gamma)$.

23. *If* $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2$ *then* $\Gamma_1 \vdash_{\mathrm{N}} N_1$, $\Gamma_2 \vdash_{\mathrm{N}} N_2$, $\forall o@l : \langle O \rangle \in \Gamma_1$ *where* $l \in \mathit{locs}(N_2)$. $o@l : \langle O \rangle \in \Gamma_2 \wedge l \notin \mathit{locs}(N_1)$, $\forall o@l : \langle O \rangle \in \Gamma_2$ *where* $l \in \mathit{locs}(N_1)$. $o@l : \langle O \rangle \in \Gamma_1 \wedge l \notin \mathit{locs}(N_2)$ *and* $\neg(o@l : \langle O_1 \rangle \in \Gamma_1 \wedge o@l : \langle O_2 \rangle \in \Gamma_2 \wedge O_1 \neq O_2)$.

24. *If* $\Gamma \vdash_{\mathrm{N}} \boldsymbol{\nu} r\, N$ *then* $\Gamma \vdash_{\mathrm{N}} N$.

Proof. The proof follows straightforward from the definition of the type system.

□

3.3.2 Structural Congruence

The semantics of Jolie makes use of structural congruence at the behavioural and the network layer. In order to have type preservation we must also have that two structural congruent statements are typable with respect to the same environment. For behaviours we furthermore require that they also make the same updates to the environment during type checking. Below we present this formally for the behavioural layer and the network layer.

3.3.2.1 Structural Congruence at Behavioural Layer

Recall from definition 2.2 in section 2.4 that the following structural congruence rules apply at the behavioural layer:

$$\begin{aligned}(\, B_1 \mid B_2 \,) \mid B_3 &\equiv B_1 \mid (\, B_2 \mid B_3 \,) \\ 0; B &\equiv B \\ B_1 \mid B_2 &\equiv B_2 \mid B_1 \\ B \mid 0 &\equiv B\end{aligned}$$

Two structural congruent behaviours are typable with respect to the same environment and they make the same updates to the environment during type checking:

Lemma 3.10 (Structural Congruence for Behaviours)

Let $\Gamma \vdash B_1 \triangleright \Gamma'$
If $B_1 \equiv B_2$
then $\Gamma \vdash B_2 \triangleright \Gamma'$

Proof.

The proof is a case analysis of all the possibilities of B_1 and B_2 according to the structural congruence rules for behaviours.

Case $(\, B_1 \mid B_2 \,) \mid B_3 \equiv B_1 \mid (\, B_2 \mid B_3 \,)$

From the case we have $\Gamma \vdash_{\mathrm{B}} (\, B_1 \mid B_2 \,) \mid B_3 \triangleright \Gamma'$. Applying the inversion lemma (lemma 3.9) we know from the form of the typing system that it is only typable by T-Par:

$$(\texttt{T-Par}) \quad \frac{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2' \quad \Gamma_3 \vdash_{\mathrm{B}} B_3 \triangleright \Gamma_3' \quad Roots(\Gamma_1' \uplus \Gamma_2') \cap Roots(\Gamma_3') = \emptyset}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{\mathrm{B}} (B_1 \mid B_2) \mid B_3 \triangleright \Gamma_1' \uplus \Gamma_2' \uplus \Gamma_3'} \tag{3.7}$$

Where $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ and $\Gamma' = \Gamma_1' \uplus \Gamma_2' \uplus \Gamma_3'$. Applying the inversion lemma on premise $\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'$ from this rule it can be seen by the form of the type system that it is only typable by rule T-Par:

$$(\texttt{T-Par}) \quad \frac{\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2' \quad Roots(\Gamma_1') \cap Roots(\Gamma_2') = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'} \tag{3.8}$$

By premise $Roots(\Gamma'_1 \uplus \Gamma'_2) \cap Roots(\Gamma'_3) = \emptyset$ we have $Roots(\Gamma'_2) \cap Roots(\Gamma'_3) = \emptyset$ as well as $Roots(\Gamma'_1) \cap Roots(\Gamma'_3) = \emptyset$. By applying premise $\Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'_2$ from 3.8 and premise $\Gamma_3 \vdash_{\mathrm{B}} B_3 \triangleright \Gamma'_3$ from 3.7 and $Roots(\Gamma'_2) \cap Roots(\Gamma'_3) = \emptyset$ on T-Par we get $\Gamma_2, \Gamma_3 \vdash_{\mathrm{B}} B_2 \mid B_3 \triangleright \Gamma'_2 \uplus \Gamma'_3$. The thesis follows by applying $\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'_1$ from 3.8 and $\Gamma_2, \Gamma_3 \vdash_{\mathrm{B}} B_2 \mid B_3 \triangleright \Gamma'_2 \uplus \Gamma'_3$ on T-Par.

Case $B_1 \mid (B_2 \mid B_3) \equiv (B_1 \mid B_2) \mid B_3$
The proof is similar to the proof for case $(B_1 \mid B_2) \mid B_3 \equiv B_1 \mid (B_2 \mid B_3)$.

Case $0; B \equiv B$
From the case we $\Gamma \vdash_{\mathrm{B}} 0; B \triangleright \Gamma'$. Applying the inversion lemma (lemma 3.9) we know from the form of the typing system that it is only typable by rule T-Seq:

$$(\texttt{T-Seq}) \quad \frac{\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma'' \qquad \Gamma'' \vdash_{\mathrm{B}} B \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \mathbf{0}; B \triangleright \Gamma'}$$

Applying the inversion lemma on premise $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma''$ from this rule it can be seen by the form of the typing system that it is only typable by rule T-Nil:

$$(\texttt{T-Nil}) \quad \frac{}{\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma}$$

From the form of this rule we know that $\Gamma'' = \Gamma$. We thereby have $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$ by the second premise of 3.3.2.1. The thesis follows by $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$.

Case $B \equiv 0; B$

From the case we have $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$. By T-Nil we have $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$. The thesis follows from applying $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$ and $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$ to rule T-Seq.

Case $B_1 | B_2 \equiv B_2 | B_1$

From the case we have $\Gamma \vdash_{\mathrm{B}} B_1 | B_2 \triangleright \Gamma'$. Applying the inversion lemma (lemma 3.9) on $\Gamma \vdash_{\mathrm{B}} B_1 | B_2 \triangleright \Gamma'$ we know that it can only be typed using T-Par:

$$(\texttt{T-Par}) \quad \frac{\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'_1 \qquad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'_2 \qquad Roots(\Gamma'_1) \cap Roots(\Gamma'_2) = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma'_1 \uplus \Gamma'_2} \tag{3.9}$$

where $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma' = \Gamma'_1 \uplus \Gamma'_2$. Since union is commutative, the thesis follows by swapping the two first premises of 3.9 and apply all premises to T-Par.

Case $B_2|B_1 \equiv B_1|B_2$
The proof is similar to the proof for case $B_1|B_2 \equiv B_2|B_1$.

Case $B \mid 0 \equiv B$

From the case we have $\Gamma \vdash_{\mathrm{B}} B \mid 0 \triangleright \Gamma'$. Applying the inversion lemma (lemma 3.9) on $\Gamma \vdash_{\mathrm{B}} B \mid 0 \triangleright \Gamma'$ we know that it can only be typed using T-Par:

$$(\texttt{T-Par}) \quad \frac{\Gamma_1 \vdash_{\mathrm{B}} B \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma_2' \quad Roots(\Gamma_1') \cap Roots(\Gamma_2') = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B \mid \mathbf{0} \triangleright \Gamma_1' \uplus \Gamma_2'} \tag{3.10}$$

Where $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma' = \Gamma_1' \uplus \Gamma_2'$. By applying the inversion lemma on premise $\Gamma_2 \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma_2'$ from 3.10 we know that it can only be typed using T-Nil:

$$(\texttt{T-Nil}) \quad \frac{}{\Gamma_2 \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma_2}$$

From the form of T-Nil we know $\Gamma_2 = \Gamma_2'$. Since Γ_1 and Γ_2 are disjoint and since we have $\Gamma_1 \vdash_{\mathrm{B}} B \triangleright \Gamma_1'$ then we must also have $\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B \triangleright \Gamma_1' \uplus \Gamma_2$.

Case $B \equiv B \mid 0$

From the case we have $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$. By T-Nil we have $\emptyset \vdash_{\mathrm{B}} \mathbf{0} \triangleright \emptyset$. The thesis follows from applying $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$ and $\emptyset \vdash_{\mathrm{B}} \mathbf{0} \triangleright \emptyset$ to T-Par.

□

3.3.2.2 Structural Congruence at Network Layer

Recall from definition 2.5 in section 2.4 that the following structural congruence rules apply:

$$\begin{aligned} (\, N_1 \mid N_2 \,) \mid N_3 &\equiv N_1 \mid (\, N_2 \mid N_3 \,) \\ N_1 \mid N_2 &\equiv N_2 \mid N_1 \\ N \mid 0 &\equiv N \\ \text{if } r \notin fn(N_2): \quad ((\boldsymbol{\nu} r\,) N_1) \mid N_2 &\equiv (\boldsymbol{\nu} r\,)(N_1 \mid N_2) \end{aligned}$$

Two structural congruent networks are typable with respect to the same environment:

Lemma 3.11 (Structural Congruence for Networks)

$$\begin{aligned}&\textit{Let } \Gamma \vdash N_1 \triangleright \Gamma'\\ &\textit{If } N_1 \equiv N_2\\ &\textit{then } \Gamma \vdash N_2 \triangleright \Gamma'\end{aligned}$$

Proof.

The proof is a case analysis of all the possibilities of N_1 and N_2 according to the structural congruence rules for networks. The proofs for the cases are similar to the proofs for their corresponding cases at the behavioural layer (see the proof for Lemma 3.10). The cases not represented at the behavioural layer are proved below:

Case $((\boldsymbol{\nu} r\,)N_1) \mid N_2 \equiv (\boldsymbol{\nu} r\,)(N_1 \mid N_2)$

From the case we have $\Gamma \vdash_{\mathrm{B}} ((\boldsymbol{\nu} r\,)N_1) \mid N_2 \triangleright \Gamma'$. Applying the inversion lemma (lemma 3.9) on $\Gamma \vdash_{\mathrm{B}} ((\boldsymbol{\nu} r\,)N_1) \mid N_2 \triangleright \Gamma'$ we know that it can only be typed using T-Network:

$$\text{(T-Network)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} (\boldsymbol{\nu} r\,)N_1 \quad \Gamma_2 \vdash_{\mathrm{N}} N_2\\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\, \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs((\boldsymbol{\nu} r\,)N_1)\\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs((\boldsymbol{\nu} r\,)N_1).\, \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2)\\ \neg(\mathtt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \mathtt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} ((\boldsymbol{\nu} r\,)N_1) \mid N_2} \tag{3.11}$$

Applying the Inversion Lemma on premise $\Gamma_1 \vdash_{\mathrm{N}} (\boldsymbol{\nu} r\,)N_1$ we know that it can only be typed using T-Restriction:

$$\text{(T-Restriction)}\quad \frac{\Gamma \vdash_{\mathrm{N}} N_1}{\Gamma_1 \vdash_{\mathrm{N}} (\boldsymbol{\nu} r\,)N_1} \tag{3.12}$$

Recall that we write $locs(N)$ for the set of locations which services in a network N are deployed at and that $locs(N)$ is recursively defined:

$$\begin{aligned} locs([S]_l) &= \{l\}\\ locs(\boldsymbol{\nu} r\; N) &= locs(N)\\ locs(N_1 \mid N_2) &= locs(N_1) \cup locs(N_2)\\ locs(\mathbf{0}) &= \emptyset\end{aligned}$$

We therefore have $locs(\boldsymbol{\nu} r\; N) = locs(N)$. By premise $\forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1$ where $l \in locs(N_2).\, \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs((\boldsymbol{\nu} r\,)N_1)$ and $\forall \mathtt{o@}l$:

$\texttt{<}O\texttt{>} \in \Gamma_2$ where $l \in locs((\boldsymbol{\nu} r)N_1).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2)$ from 3.11 we know

$$\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1)$$
$$\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2)$$

By applying premise $\Gamma \vdash_{\mathrm{N}} N_1$ from 3.12 and premise $\Gamma_2 \vdash_{\mathrm{N}} N_2$ and $\neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)$ from 3.11 and 3.13 to T-Network we get $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2$. The thesis follows from applying $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2$ to T-Restriction.

Case $(\boldsymbol{\nu} r)(N_1 \mid N_2) \equiv ((\boldsymbol{\nu} r)N_1) \mid N_2$

The proof for this case is similar to the proof for its symmetric case.

□

3.3.3 Transition Function

The type preservation property address typability of statements before and after the given statement takes a transition. Since actions are performed during the transitions in a labeled transition system, we therefore define a transition function which updates the environment with respect to the action performed.

Let B and B' be two behaviours and let μ be a label. Consider the case $B \xrightarrow{\mu} B'$, and let B be typed with respect to Γ and B' be typed with respect to Γ'. The relation between Γ and Γ' is defined as the side effect of μ. It is described in the function *sideEffect*:

Definition 3.12 (*sideEffect*)

$$
sideEffect(\mu,\Gamma) = \begin{cases}
\Gamma & \text{if } \mu = \texttt{x = } e \text{ and } \Gamma \vdash e : T_e \\
 & \text{and } x : T_x \in \Gamma \text{ and } bt(T_e) = bt(T_x) \\
upd(\Gamma, x, bt(T_e)) & \text{if } \mu = \texttt{x = } e \text{ and } \Gamma \vdash e : T_e \\
 & \text{and } x \notin \Gamma \\
\Gamma & \text{if } \mu = (r, \texttt{o@}l)?x \text{ and } \texttt{o@}l : \texttt{<}T_o, T_i\texttt{>} \in \Gamma \\
 & \text{and } x : T_x \in \Gamma \text{ and } T_i \leq T_x \\
upd(\Gamma, x, T_i) & \text{if } \mu = (r, \texttt{o@}l)?x \text{ and } \texttt{o@}l : \texttt{<}T_o, T_i\texttt{>} \in \Gamma \\
 & \text{and } x \notin \Gamma \\
\Gamma & \text{if } \mu = r : \texttt{o(x)} \text{ and } x : T_x \in \Gamma \\
 & \text{and } (\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma \vee \texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma) \\
 & \text{and } T_i \leq T_x \\
upd(\Gamma, x, T_i) & \text{if } \mu = r : \texttt{o(x)} \text{ and } x \notin \Gamma \\
 & \text{and } (\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma \vee \texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma) \\
\Gamma & \text{if } \mu = \mathsf{read}\ t \\
\Gamma & \text{if } \mu = \boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)} \text{ and } \Gamma \vdash e : T_e \\
 & \text{and } (\texttt{o@}l : \texttt{<}T_o\texttt{>} \in \Gamma \vee \texttt{o@}l : \texttt{<}T_o, T_i\texttt{>} \in \Gamma) \\
 & \text{and } T_e \leq T_o \\
\Gamma & \text{if } \mu = (r, \texttt{o})!\, x \text{ and } x : T_x \in \Gamma \\
 & \text{and } \texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma) \text{ and } T_x \leq T_o \\
\text{undefined} & \text{otherwise}
\end{cases}
\tag{3.13}
$$

For understanding the first case recall that we write $\Gamma \vdash e : T$ when the result of the evaluation of expression e under type environment Γ has minimal type T and that we write $bt(T)$ for root node of a type T omitting the type information of the children.

For understanding the second case recall also that the function *upd* takes as input an typing environment Γ, a variable x and a type T_x. It updates the part x of Γ with type T_x. The form of the output differs regarding the form of Γ: If the root of variable x is in Γ, then the root is bound to a type tree representing x with type T_x and in which all missing predecessors of x are assigned type `void`. This type tree is built by *addPath*. If the root of x is not in Γ and if x is not the root of itself, a similar type tree is build up by *addPath* but instead of building of an existing type tree, a new consisting of a single node with type `void` is used. If x is an edge to a root node, and if it is not in Γ, the type T_x is

assigned directly to x without use of help functions.

$$upd(\Gamma, x, T_x) = \begin{cases} \Gamma[r(x) \mapsto addPath(T_{r(x)}, x, T_x)] & \text{if } r(x) : T_{r(x)} \in \Gamma \\ \Gamma[r(x) \mapsto addPath(\texttt{void}, x, T_x)] & \text{if } r(x) \notin \Gamma \wedge r(x) \neq x \\ \Gamma[x \mapsto T_x] & \text{if } r(x) \notin \Gamma \wedge r(x) = x \end{cases} \tag{3.14}$$

Recall that when a variable is given a value, where one or more predecessors of the variable's path haven't been given any value, the predecessors get the type `void`. The function *addPath* takes a path x and two types T and T_x. It steps through x from its leaf a and builds up the type tree for x, where the leaf of x gets type T_x and the missing predecessors get type `void`. When it reaches an existing predecessor of x it connects the builded tree to the predecessor using *addChild*.

$$addPath(T, x, T_x) = \begin{cases} addPath(T, p(x) : \texttt{void}\{.a : T_x\}) & \text{if } p(x) \notin T \wedge x = p(x).a \\ addChild(T, x : T_x) & \text{if } p(x) \in T \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.15}$$

where $r(x)$ denotes the root edge of path x. The function *addChild* takes a type T and a type binding $x : T_x$ consisting of a path x and a type T_x. It adds path x with type T_x to path $p(x)$ in T.

$$addChild(T, x : T_x) = T' \text{ where } \forall x' \in T \,.\, x' \in T' \text{ and } x : T_x \in T'$$

The function is undefined if

$$x : T'_x \in T \text{ where } T'_x \neq T_x$$

and if

$$p(x) \notin T \ \wedge \ p(x) \neq r(x)$$

where we let $p(x)$ denote the path to the parent of the node pointed to by x. If x is root $p(x) = \bot$.

The rest of the cases are understandable with the previous in memory.

The labels in the definition of *sideEffect* differs into two kinds: Those which perform a reading action and those which perform a writing action. It can be seen from the first six cases in *sideEffect* that during a transition labelled with a writing action, the environment is updated with type information for the variable in the label, unless the change did not change the type of the variable. The next three cases shows that the environment is unchanged when a transition labelled with a reading action is taken.

3.3.4 Type Preservation

The property that if a well-typed statement takes a transition then the resulting statement is also well typed, is called type preservation. Recall that the type judgement for behaviours is a Hoare triple consisting of a initial environment, a behaviour and a resulting environment, where the resulting environment is the initial environment updated with the changes performed during the type checking of the behaviour. The type preservation property therefore requires for behaviours that the resulting environment for a behaviour after it takes a transition is equal to the resulting environment for the behaviour before it takes the transition. Below we present type preservation theorems for each of the layers in Jolie.

3.3.4.1 Type Preservation at the Behavioural Layer

Consider a well-typed behaviour B typed with respect to an environment Γ. Let the type checking of B update Γ to Γ'. Assume there exists a behaviour B' such that $B \xrightarrow{\mu} B'$. Then there exists an environment Γ'' which is an update of Γ with respect to the side effect of label μ, and B' is typable with respect to Γ'' which is updated to Γ' during the type checking of B:

Theorem 3.13 (Type Preservation for Behaviours)

$$
\begin{aligned}
&\textit{If } \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \\
&\textit{and } B \xrightarrow{\mu} B' \\
&\textit{then } \Gamma'' \vdash_{\mathrm{B}} B \triangleright \Gamma' \\
&\textit{where } \Gamma'' = \textit{sideEffect}(\mu, \Gamma)
\end{aligned}
$$

Proof.

Assume $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$ and $B \xrightarrow{\mu} B'$. The proof is done by induction on the derivation of $B \xrightarrow{\mu} B'$.

Case B-If-Then

In this case, we know $B = \texttt{if}(e)\ B_1\ \texttt{else}\ B_2$ for an expression e and some behaviours B_1 and B_2:

$$
(\texttt{B-If-Then}) \quad \frac{e(t) = true}{\texttt{if}(e)\ B_1\ \texttt{else}\ B_2 \xrightarrow{\textbf{read}\ t} B_1}
$$

We also know that $\mu = \mathsf{read}\ t$ and $B' = B_1$. Since $\Gamma'' = \mathit{sideEffect}(\mathsf{read}\ t, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12) we are going to prove that $\Gamma \vdash B' \triangleright \Gamma'$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{if}(e)\ B_1\ \texttt{else}\ \ B_2 \triangleright \Gamma'$ we know that it can only be typed by rule T-If-Then-Else:

$$(\texttt{T-If-Then-Else}) \quad \frac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \texttt{if}(e)\ B_1\ \texttt{else}\ \ B_2 \triangleright \Gamma'}$$

The thesis follows by premise $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'$.

Case B-If-Else The proof is similar to case B-If-Then.

Case B-Par

In this case, we know $B = B_1 \mid B_2$ for some behaviours B_1 and B_2:

$$(\texttt{B-Par}) \quad \frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \mid B_2 \xrightarrow{\mu} B_1' \mid B_2}$$

We also know $B' = B_1' \mid B_2$.

Applying the Inversion Lemma (Lemma 3.9) on $\Gamma \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma'$ we know that it can only be typed by rule T-Par where $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma' = \Gamma_1' \uplus \Gamma_2'$:

$$(\texttt{T-Par}) \quad \frac{\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2' \quad \mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'} \tag{3.16}$$

By applying the induction hypothesis to the premise of B-Par, $B_1 \xrightarrow{\mu} B_1'$, and premise, $\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1'$, of T-Par we get $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma_1'$where $\Gamma_1'' = \mathit{sideEffect}(\mu, \Gamma_1)$. Since it is not possible to remove variables from an environment by the form of the type system and *SideEffect* we know by $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma_1'$ from the induction hypothesis and by $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$ from 3.16 that $\mathit{Roots}(\Gamma_1'') \cap \mathit{Roots}(\Gamma_2) = \emptyset$. Since it is not possible to alter operations by the form of the type system and *SideEffect* we also know

$$\Gamma_1'' \cap \Gamma_2 = \emptyset \tag{3.17}$$

From the conclusion of 3.16 we know $\Gamma_1' \uplus \Gamma_2'$. Since we know $\Gamma_1'' \cap \Gamma_2 = \emptyset$ and $\Gamma_1' \uplus \Gamma_2'$ then with $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma_1'$ from the induction hypothesis and $\Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2'$ and $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$ from T-Par on B, we can now apply T-Par on B':

$$(\texttt{T-Par}) \quad \frac{\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2' \quad \mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset}{\Gamma_1'', \Gamma_2 \vdash_{\mathrm{B}} B_1' \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'}$$

In order for the thesis to follow from the conclusion of this rule, we must have $\Gamma_1'' \uplus \Gamma_2 = \mathit{sideEffect}(\mu, \Gamma)$. In order to prove that, we first prove $\mathit{Roots}(\mu) \subseteq \mathit{Keys}(\mathit{sideEffect}(\mu, \Gamma))$, where $\mathit{Keys}(\Gamma_k)$ denotes the keys in Γ_k for an environment Γ_k.

Let $\mathit{Roots}(e)$ be the set of roots of variables in an expression e, and let $\mathit{Roots}(\mu)$ be the set of roots of variables in a label μ including roots of variables from any expressions in μ.

Let a well-typed behaviour B typed under an environment Γ take a transition labelled μ. Let the transition update Γ to Γ''. The set of variable roots in μ is a subset of the set of variable roots in Γ''.

Lemma 3.14

If $\Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma'$ *and* $B \xrightarrow{\mu} B'$
then $\mathit{Roots}(\mu) \subseteq \mathit{Keys}(\mathit{sideEffect}(\mu, \Gamma))$

Proof.

The proof is a case analysis over the labels.

Case $\mu = \mathtt{x = } e$

From the case we know $\mathit{Roots}(\mu) = r(x) \cup \mathit{Roots}(e)$. From the form of *SideEffect* we know that in order for *SideEffect* to return, there exists two possibilities for a transition taken with this label:

Subcase $\mathit{SideEffect}(\mathtt{x = } e, \Gamma) = \Gamma$

From the for of *sideEffect* we know $\Gamma \vdash e : T_e$ and $x : T_x \in \Gamma$. By $\Gamma \vdash e : T_e$ we know $\mathit{Roots}(e) \subseteq \mathit{Keys}(\mathit{sideEffect}(\mathtt{x = } e, \Gamma))$. By $x : T_x \in \Gamma$ we know $r(x) \in \Gamma$ from the form of *sideEffect*. By $r(x) \in \Gamma$ and $\mathit{SideEffect}(\mathtt{x = } e, \Gamma) = \Gamma$ we know $r(x) \in \mathit{Keys}(\mathit{sideEffect}(\mathtt{x = } e, \Gamma))$.

Subcase $\mathit{SideEffect}(\mathtt{x = } e, \Gamma) = \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))$

From the for of *sideEffect* we know $\Gamma \vdash e : T_e$. By $\Gamma \vdash e : T_e$ we know $\mathit{Roots}(e) \subseteq \mathit{Keys}(\mathit{sideEffect}(\mathtt{x = } e, \Gamma))$. By the form of *upd* we know $r(x) \in \mathit{Keys}(\mathit{sideEffect}(\mathtt{x = } e, \Gamma))$.

Case $\mu = (r, \mathtt{o@l})?x$

The proof is similar to the proof for case $\mu = \mathtt{x = } e$.

Case $\mu = r : \mathtt{o(x)}$

The proof is similar to the proof for case $\mu = \mathtt{x = } e$.

Case $\mu = \textsf{read}\ t$

From the case we know $\mathit{Roots}(\mu) = \emptyset$. The thesis follows immediately.

Case $\mu = \boldsymbol{\nu} r\ \texttt{o@l(e)}$

From the case we know $\mathit{Roots}(\mu) = \mathit{Roots}(e)$ and $\mathit{sideEffect}(\boldsymbol{\nu} r \texttt{o@l(e)}, \Gamma) = \Gamma$. From the for of *sideEffect* we know $\Gamma \vdash e : T_e$. By $\Gamma \vdash e : T_e$ and $\mathit{sideEffect}(\boldsymbol{\nu} r \texttt{o@l(e)}, \Gamma) = \Gamma$ we know $\mathit{Roots}(e) \subseteq \mathit{Keys}(\mathit{sideEffect}(\boldsymbol{\nu} r \texttt{o@l(e)}, \Gamma))$.

Case $\mu = (r, \texttt{o})!\, x$

From the case we know $\mathit{Roots}(\mu) = r(x)$ and $\mathit{sideEffect}((r, \texttt{o})!\, x, \Gamma) = \Gamma$. From the for of *sideEffect* we know $x : T_x \in \Gamma$. By $x : T_x \in \Gamma$ we know $r(x) : T_x \in \Gamma$ from the form of *sideEffect*. From $r(x) : T_x \in \Gamma$ and $\mathit{sideEffect}(\boldsymbol{\nu} r \texttt{o@l(e)}, \Gamma) = \Gamma$ we know $r(x) \in \mathit{Keys}(\mathit{sideEffect}(\boldsymbol{\nu} r \texttt{o@l(e)}, \Gamma))$.

□

By $B_1 \xrightarrow{\mu} B_1'$ from the case, $\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1'$ from 3.16 and Lemma 3.14 we know $\mathit{Roots}(\mu) \subseteq \mathit{Keys}(\mathit{SideEffect}(\mu, \Gamma_1))$. Since it is not possible to remove variables from an environment by the form of the type system and *SideEffect* we know by $\mathit{Roots}(\mu) \subseteq \mathit{SideEffect}(\mu, \Gamma_1)$ and by $\mathit{SideEffect}(\mu, \Gamma_1) \vdash_{\mathrm{B}} B_1' \triangleright \Gamma_1'$ from the induction hypothesis and by $\mathit{Roots}(\Gamma_1') \cap \mathit{Roots}(\Gamma_2') = \emptyset$ from 3.16 that $\mathit{Roots}(\mu) \cap \Gamma_2' = \emptyset$. Since it is not possible to remove variables from an environment by the form of the type system and *SideEffect* we know by $\mathit{Roots}(\mu) \cap \Gamma_2' = \emptyset$ and by $\Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2'$ from 3.16 that

$$\mathit{Roots}(\mu) \cap \Gamma_2 = \emptyset \tag{3.18}$$

If two environments Γ_1 and Γ_2 are disjoint and if the effect of executing a label μ in the context Γ_1, updates Γ_1 to Γ_1'' and if the set of roots of the variables in μ are disjoint to Γ_2 then the disjoint union of Γ_2 to Γ_1 does not influence the updates performed by executing μ.

Lemma 3.15

If $\mathit{SideEffect}(\mu, \Gamma_1) = \Gamma_1''$
and $\Gamma_1 \cap \Gamma_2 = \emptyset$
and $\mathit{Roots}(\mu) \cap \Gamma_2 = \emptyset$
then $\mathit{SideEffect}(\mu, (\Gamma_1 \uplus \Gamma_2)) = \mathit{SideEffect}(\mu, \Gamma_1) \uplus \Gamma_2$

Proof.

The proof is a case analysis over the labels.

Case $\mu = \mathtt{x} = e$

Since we know $\mathit{Roots}(\mathtt{x} = e) \cap \Gamma_2 = \emptyset$ we know that $x \notin \Gamma_2$ and that $\mathit{Roots}(e) \cap \Gamma_2 = \emptyset$. We therefore know that since we have $\mathit{SideEffect}(\mathtt{x} = e, \Gamma_1) = \Gamma_1'$ then union Γ_2 to the input, does not influence the branch chosen in *SideEffect*.

The rest of the cases are similar.

□

Recall we in 3.17 have $\mathit{sideEffect}(\mu, \Gamma_1) \cap \Gamma_2 = \emptyset$ and that $\Gamma = \Gamma_1, \Gamma_2$. The thesis follows by $\mathit{sideEffect}(\mu, \Gamma_1) = \Gamma_1''$ from the induction hypothesis and by $\Gamma_1 \cap \Gamma_2$, 3.18, $\mathit{sideEffect}(\mu, \Gamma_1) \cap \Gamma_2 = \emptyset$ and Lemma 3.15.

Case B-Seq

In this case, we know $B = B_1; B_2$ for some behaviours B_1 and B_2:

$$(\texttt{B-Seq}) \quad \frac{B_1 \xrightarrow{\mu} B_1'}{B_1;B_2 \xrightarrow{\mu} B_1';B_2}$$

We also know $B' = B_1'; B_2$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} B_1; B_2 \triangleright \Gamma'$ we know that it can only be typed by rule T-Seq:

$$(\texttt{T-Seq}) \quad \frac{\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad \Gamma''' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} B_1;B_2 \triangleright \Gamma'}$$

By applying the induction hypothesis on the premise $B_1 \xrightarrow{\mu} B_1'$ of B-Seq and premise $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'''$ of T-Seq, we get $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma'''$ where $\Gamma_1'' = \mathit{sideEffect}(\mu, \Gamma)$.

We can now apply T-Seq on B' with the two premises, $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma'''$ from the induction hypothesis and $\Gamma''' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'$ from T-seq used on B:

$$(\texttt{T-Seq}) \quad \frac{\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma''' \quad \Gamma''' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'}{\Gamma_1'' \vdash_{\mathrm{B}} B_1';B_2 \triangleright \Gamma'}$$

In order for the thesis to follow from the conclusion of T-Seq applied on B' we must have that $\Gamma_1'' = \Gamma''$. Since $\Gamma_1'' = \mathit{sideEffect}(\mu, \Gamma)$ and $\Gamma'' = \mathit{sideEffect}(\mu, \Gamma)$ then $\Gamma_1'' = \Gamma''$.

Case B-Iteration

In this case, we know $B = \texttt{while(}e\texttt{)}\,\{B_1\}$ for an expression e and a behaviour B_1:

$$(\texttt{B-Iteration}) \quad \frac{e(t)=true}{\texttt{while}(e)\,\{B_1\} \xrightarrow{\textsf{read}\ t} B_1;\texttt{while}(e)\,\{B_1\}}$$

We also know that $\mu = \textsf{read}\ t$ and $B' = B_1;\texttt{while}(e)\,\{B_1\}$. Since $\Gamma'' = \mathit{sideEffect}(\textsf{read}\ t, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12) we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma'$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma'$ we know that it can only be typed by rule T-While:

$$(\texttt{T-While}) \quad \frac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma}$$

Notice that B is only typable for cases where $\Gamma' = \Gamma$. Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. By premise $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma$ and the conclusion $\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma$ in T-While applied on B we can now apply T-Seq on B':

$$(\texttt{T-Seq}) \quad \frac{\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma \quad \Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma}{\Gamma \vdash_{\mathrm{B}} B_1;\texttt{while}(e)\,\{B_1\} \triangleright \Gamma}$$

The thesis follows from the conclusion of T-Seq applied on B'.

Case B-No-Iteration

If the last rule in the derivation sequence is B-No-Iteration, then from the form of this rule, we see that $B = \texttt{while}(e)\,\{B_1\}$ for an expression e and a behaviour B_1:

$$(\texttt{B-No-Iteration}) \quad \frac{e(t)=false}{\texttt{while}(e)\,\{B_1\} \xrightarrow{\textsf{read}\ t} \mathbf{0}}$$

We also know that $\mu = \textsf{read}\ t$ and $B' = \mathbf{0};\texttt{while}(e)\,\{B_1\}$. Since $\Gamma'' = \mathit{sideEffect}(\textsf{read}\ t, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12) we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma'$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma'$ we know that it can only be typed by rule T-While:

$$(\texttt{T-While}) \quad \frac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{while}(e)\,\{B_1\} \triangleright \Gamma}$$

From the conclusion of this rule we have that $\Gamma' = \Gamma$. Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Case B-Assign

In this case, we know $B = \texttt{x = }e$ for an expression e and a variable x:

$$(\texttt{B-Assign}) \quad \texttt{x = }e \xrightarrow{\texttt{x = }e} \mathbf{0}$$

We also know that $\mu = \texttt{x = } e$ and $B' = \mathbf{0}$.

Applying the Inversion Lemma (lemma 3.9) on $\Gamma \vdash_{\mathrm{B}} \texttt{x = } e \triangleright \Gamma'$ we know that it can be typed by two different rules. We therefore consider each case:

Subcase T-Assign-New

If $\Gamma \vdash_{\mathrm{B}} \texttt{x = } e \triangleright \Gamma'$ is typed using T-Assign-New, then we see from the form of this rule that $\Gamma' = \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))$:

$$(\texttt{T-Assign-New}) \quad \frac{\Gamma \vdash e : T_e \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{x = } e \triangleright \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))}$$

By premise $\Gamma \vdash e : T_e$ and $x \notin \Gamma$ from T-Assign-New applied on B and by label $\texttt{x = } e$ we have that $\Gamma'' = \mathit{sideEffect}(\texttt{x = } e, \Gamma) = \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\mathit{upd}(\Gamma, x, \mathit{bt}(T_e)) \vdash_{\mathrm{B}} P' \triangleright \mathit{upd}(\Gamma, x, \mathit{bt}(T_e))$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Subcase T-Assign-Exists

If $\Gamma \vdash_{\mathrm{B}} \texttt{x = } e \triangleright \Gamma'$ is typed using T-Assign-Exists, then we see from the form of this rule that $\Gamma' = \Gamma$:

$$(\texttt{T-Assign-Exists}) \quad \frac{\Gamma \vdash e : T_e \quad x : T_x \in \Gamma \quad \mathit{bt}(T_e) = \mathit{bt}(T_x)}{\Gamma \vdash_{\mathrm{B}} \texttt{x = } e \triangleright \Gamma}$$

We are therefore going to prove that $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma$. By premise $\Gamma \vdash e : T_e$, $x : T_x \in \Gamma$ and $\mathit{bt}(T_e) = \mathit{bt}(T_x)$ from T-Assign-Exists applied on B and by label $\texttt{x = } e$ we have that $\mathit{sideEffect}(\texttt{x = } e, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Case B-Notification

In this case, we know $B = \texttt{o@}l(e)$ for an expression e and an operation $\texttt{o}$ at a location l:

$$(\texttt{B-Notification}) \quad \texttt{o@}l\texttt{(e)} \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}} \mathbf{0}$$

We also know that $\mu = \boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ and $B' = \mathbf{0}$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l(e) \triangleright \Gamma'$ we know that it can only be typed by rule T-Notification:

$$(\texttt{T-Notification}) \quad \frac{\texttt{o@}l : \langle T_o \rangle \in \Gamma \quad \Gamma \vdash e : T_e \quad T_e \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)} \triangleright \Gamma}$$

From the conclusion of this rule we know that $\Gamma' = \Gamma$. Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma$. By the premises of this rule together with label $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ we have that $\Gamma'' = \mathit{sideEffect}(\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Case B-SolResp

If the last rule in the derivation sequence is B-SolResp, then from the form of this rule, we see that $B = \texttt{o@}l\texttt{(e)(x)}$ for an expression e, a variable x,an operation $\texttt{o}$ at a location l:

$$(\texttt{B-SolResp}) \quad \texttt{o@}l\texttt{(e)(x)} \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x})$$

We also know that $\mu = \boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ and $B' = \mathsf{Wait}(r, \texttt{o@}l, \texttt{x})$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma'$ we know that it can be typed by two rules. We therefore consider both cases:

Subcase T-SolResp-New

If $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma'$ is typed using T-SolicitResponse-New then we see from the form of this rule that $\Gamma' = \mathit{upd}(\Gamma, x, T_i)$:

$$(\texttt{T-SolResp-New}) \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \mathit{upd}(\Gamma, x, T_i)}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$.

By premise $\texttt{o@}l : \langle T_o, T_i\rangle \in \Gamma$, $\Gamma \vdash e : T_e$ and $T_e \leq T_o$ of this rule and by label $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ we have that $\Gamma'' = \mathit{sideEffect}(\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$. The thesis follows from applying rule T-Wait-New on $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \mathit{upd}(\Gamma, x, T_i)$ by premise $\texttt{o@}l : \langle T_o, T_i\rangle \in \Gamma$ and $x \notin \Gamma$ from T-SolResp-New applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma'$:

$$(\texttt{T-Wait-New}) \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \mathit{upd}(\Gamma, x, T_i)}$$

Subcase T-SolicitResponse-Exists

If $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma'$ is typed using T-SolicitResponse-Exists then we see from the form of this rule that $\Gamma' = \Gamma$:

$$(\texttt{T-SolResp-Exists}) \quad \frac{\texttt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma$.

By premise $\texttt{o@}l : \langle T_o, T_i\rangle \in \Gamma, \Gamma \vdash e : T_e$ and $T_e \leq T_o$ of this rule and by label $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}$ we have that $\Gamma'' = \mathit{sideEffect}(\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(e)}, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying rule T-Wait-Exists on $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \Gamma$ by premise $\texttt{o@}l : \langle T_o, T_i\rangle \in \Gamma$, $x : T_x \in \Gamma$ and $T_i \leq T_x$ from T-SolResp-Exists applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o@}l\texttt{(e)(x)} \triangleright \Gamma'$:

$$(\texttt{T-Wait-Exists})\frac{\texttt{o@}l{:}\texttt{<}T_o,T_i\texttt{>} \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r,\texttt{o@}l,\texttt{x}) \triangleright \Gamma}$$

Case B-OneWay

If the last rule in the derivation sequence is B-OneWay, then from the form of this rule, we see that $B = \texttt{o(x)}$ for an operation o and a variable x:

$$(\texttt{B-OneWay}) \quad \texttt{o(x)} \xrightarrow{r:\texttt{o(x)}} \mathbf{0}$$

We also know that $\mu = r : \texttt{o(x)}$ and $B' = \mathbf{0}$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \Gamma'$ we know that it can be typed by two different rules. We therefore consider both cases:

Subcase T-OneWay-New

If $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \Gamma'$ is typed using T-OneWay-New, then we see from the form of this rule that $\Gamma' = \mathit{upd}(\Gamma, x, T_i)$:

$$(\texttt{T-OneWay-New}) \quad \frac{\texttt{o:<}T_i\texttt{>} \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \mathit{upd}(\Gamma,x,T_i)}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$.

By premise $x \notin \Gamma$ and $\texttt{o} : \texttt{<}T_i\texttt{>} \in \Gamma$ in T-OneWay-New applied on B and by label $r : \texttt{o(x)}$ we have that $\Gamma'' = \mathit{sideEffect}(r : \texttt{o(x)}, \Gamma) = \mathit{upd}(\Gamma, x, T_i)$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Subcase T-OneWay-Exists

If $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \Gamma'$ is typed using T-OneWay-Exists, then we see from the form of this rule that $\Gamma' = \Gamma$:

$$(\texttt{T-OneWay-Exists}) \quad \frac{\texttt{o:<}T_i\texttt{>} \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)} \triangleright \Gamma}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma$. By the premises of T-OneWayExists applied on B and by label $r : \texttt{o(x)}$ we know that $\Gamma'' = \mathit{sideEffect}(r : \texttt{o(x)}, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying T-Nil on $B' = \mathbf{0}$.

Case B-ReqResp

If the last rule in the derivation sequence is B-ReqResp, then from the form of this rule, we see that $B = \texttt{o(x)(x')\{}B_1\texttt{\}}$ for an operation o, a behaviour B_1 and two variables x and x':

$$(\texttt{B-ReqResp}) \quad \texttt{o(x)(x')}\{B_1\} \xrightarrow{r:\texttt{o(x)}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1)$$

We also know that $\mu = r : \texttt{o(x)}$ and $B' = \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1)$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$ we know that it can be typed by two different rules. We therefore consider each case:

Subcase T-ReqResp-New

If $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$ is typed using T-ReqResp-New then we see from the form of this rule that $\Gamma' = \Gamma'''$:

$$(\texttt{T-ReqResp-New}) \frac{\texttt{o}:\langle T_i, T_o\rangle \in \Gamma \quad x \notin \Gamma \quad upd(\Gamma, x, T_i) \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad x' : T_{x'} \in \Gamma''' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'''}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma'''$. By premise $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma$ and $x \notin \Gamma$ in T-ReqResp-New applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$ and by label $r : \texttt{o(x)}$ we have that $\Gamma'' = \mathit{sideEffect}(r : \texttt{o(x)}, \Gamma) = \mathit{upd}(\Gamma, x, T_i)$ by the definition of *sideEffect* (3.12). Therefore we are going to prove $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} B' \triangleright \Gamma'''$.

Since *sideEffect* does not change operations we have by premise $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma$ from T-ReqResp-New applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$ that $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma''$. The thesis follows from applying rule T-Exec on $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1) \triangleright \Gamma'''$ by $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma''$ and premise $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'''$, $x' : T_{x'} \in \Gamma'''$ and $T_{x'}) \leq T_o$ from T-ReqResp-New applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'''$:

$$(\texttt{T-Exec}) \quad \frac{\texttt{o}:\langle T_i, T_o\rangle \in \Gamma'' \quad \Gamma'' \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad x' : T_{x'} \in \Gamma''' \quad T_{x'} \leq T_o}{\Gamma'' \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1) \triangleright \Gamma'''}$$

Subcase T-ReqResp-Exists

If $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$ is typed using T-ReqResp-Exists then we see from the form of this rule that $\Gamma' = \Gamma'''$:

$$(\texttt{T-ReqResp-Exists}) \quad \frac{\texttt{o}:\langle T_i, T_o\rangle \in \Gamma \quad x : T_x \in \Gamma \quad T_i \leq T_x \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad x' : T_{x'} \in \Gamma''' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'''}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1) \triangleright \Gamma'''$.

By premise $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma$, $x : T_x \in \Gamma$ and $T_i \leq T_x$ of this rule and by label $r : \texttt{o(x)}$ we have that $\Gamma'' = \mathit{sideEffect}(r : \texttt{o(x)}, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1) \triangleright \Gamma'''$. The thesis follows from applying rule T-Exec on $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x'}, B_1) \triangleright \Gamma'''$ by premise $\texttt{o} : \langle T_i, T_o\rangle \in \Gamma$, $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'''$, $x' : T_{x'} \in \Gamma'''$ and $T_{x'} \leq T_o$ from T-ReqResp-Exists applied on $\Gamma \vdash_{\mathrm{B}} \texttt{o(x)(x')}\{B_1\} \triangleright \Gamma'$:

$$(\texttt{T-Exec}) \quad \frac{\texttt{o}{:}\texttt{<}T_i,T_o\texttt{>} \in \Gamma \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad x' : T_{x'} \in \Gamma''' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r,\texttt{o},\texttt{x'},B_1) \triangleright \Gamma'''}$$

Case B-Exec

In this case, we know $B = \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1)$ for a channel r, an operation $\texttt{o}$, a behaviour B_1 and a variable x:

$$(\texttt{B-Exec}) \quad \frac{B_1 \xrightarrow{\mu} B_1'}{\mathsf{Exec}(r,\texttt{o},\texttt{x},B_1) \xrightarrow{\mu} \mathsf{Exec}(r,\texttt{o},\texttt{x},B_1')}$$

We also know that $B' = \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1')$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1) \triangleright \Gamma'$ we know that it can only be typed by T-Exec:

$$(\texttt{T-Exec}) \quad \frac{\texttt{o}{:}\texttt{<}T_i,T_o\texttt{>} \in \Gamma \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma''' \quad x : N_x \in \Gamma''' \quad T_x \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r,\texttt{o},\texttt{x},B_1) \triangleright \Gamma'''}$$

From the conclusion of this rule we have that $\Gamma' = \Gamma'''$. Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1') \triangleright \Gamma'''$.

By applying the induction hypothesis on premise $B_1 \xrightarrow{\mu} B_1'$ from B-Exec and on premise $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'''$ from T-Exec applied on $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1) \triangleright \Gamma'''$, we get $\Gamma_1'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma'''$ where $\Gamma_1'' = \mathit{sideEffect}(\mu, \Gamma)$. From B-Exec we know that the label for the premise and for the conclusion is the same. We therefore have $\Gamma'' = \Gamma_1''$.

Since *sideEffect* does not change operations we have by premise $\texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma$ from T-Exec applied on $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1) \triangleright \Gamma'''$ that $\texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma''$. The thesis follows from applying T-Exec on $\Gamma'' \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1') \triangleright \Gamma'''$ by $\texttt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma''$, $\Gamma'' \vdash_{\mathrm{B}} B_1' \triangleright \Gamma'''$ from the induction hypothesis and premise $x : N_x \in \Gamma'''$ and $T_x \leq T_o$ from T-Exec applied on $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, B_1) \triangleright \Gamma'''$.

Case B-End-Exec

If the last rule in the derivation sequence is B-End-Exec, then from the form of this rule, we see that $B = \mathsf{Exec}(r, \texttt{o}, \texttt{x}, \mathbf{0})$ for a channel r, an operation $\texttt{o}$ and a variable x:

$$(\texttt{B-End-Exec}) \quad \mathsf{Exec}(r, \texttt{o}, \texttt{x}, \mathbf{0}) \xrightarrow{(r,\texttt{o})!\,x} \mathbf{0}$$

We also know that $\mu = (r, \texttt{o})!\,x$ and that $B' = \mathbf{0}$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, \mathbf{0}) \triangleright \Gamma'$ we know that it can only be typed by T-Exec:

$$(\texttt{T-Exec}) \quad \frac{\texttt{o}:<T_i,T_o> \in \Gamma \quad \Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma''' \quad x:T_x \in \Gamma''' \quad T_x \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r,\texttt{o},\texttt{x},\mathbf{0}) \triangleright \Gamma'''}$$

By applying T-Nil on premise $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma'''$ from this rule we know $\Gamma''' = \Gamma$. Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$. By premise $\texttt{o} : <T_i, T_o> \in \Gamma$, $x : T_x \in \Gamma$ and $T_x \leq T_o$ of T-Exec applied on $\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r, \texttt{o}, \texttt{x}, \mathbf{0}) \triangleright \Gamma$ and by label $(r, \texttt{o})!\, x$ we have that $\Gamma'' = \mathit{sideEffect}((r, \texttt{o})!\, x, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$. The thesis follows from applying rule T-Nil on $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$.

Case B-Wait

In this case, we know $B = \mathsf{Wait}(r, \texttt{o@}l, \texttt{x})$ for a channel r, a variable x and an operation $\texttt{o}$ at a location l:

$$(\texttt{B-Wait})\mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \xrightarrow{(r,\texttt{o@}l)?x} \mathbf{0}$$

We also know that $\mu = (r, \texttt{o@}l)?x$ and $B' = \mathbf{0}$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \Gamma'$ we know that it can be typed by two rules. We therefore consider both cases:

Subcase T-Wait-New

If $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \Gamma'$ is typed using T-Wait-New then we see from the form of this rule that $\Gamma' = \mathit{upd}(\Gamma, x, T_i)$:

$$(\texttt{T-Wait-New}) \frac{\texttt{o@}l:<T_o,T_i> \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r,\texttt{o@}l,\texttt{x}) \triangleright \mathit{upd}(\Gamma,x,T_i)}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$.

By premise $\texttt{o@}l : <T_o, T_i> \in \Gamma$ and $x \notin \Gamma$ of this rule and by label $(r, \texttt{o@}l)?x$ we have that $\Gamma'' = \mathit{sideEffect}((r, \texttt{o@}l)?x, \Gamma) = \mathit{upd}(\Gamma, x, T_i)$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} B' \triangleright \mathit{upd}(\Gamma, x, T_i)$. The thesis follows from applying rule T-Nil on $\mathit{upd}(\Gamma, x, T_i) \vdash_{\mathrm{B}} \mathbf{0} \triangleright \mathit{upd}(\Gamma, x, T_i)$.

Subcase T-Wait-Exists

If $\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r, \texttt{o@}l, \texttt{x}) \triangleright \Gamma'$ is typed using T-Wait-Exists then we see from the form of this rule that $\Gamma' = \Gamma$:

$$(\texttt{T-Wait-Exists}) \quad \frac{\texttt{o@}l:<T_o,T_i> \in \Gamma \quad x:T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r,\texttt{o@}l,\texttt{x}) \triangleright \Gamma}$$

Therefore we are going to prove $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma$.

By premise $\texttt{o@}l : <T_o, T_i> \in \Gamma$, $x : T_x \in \Gamma$ and $T_i \leq T_x$ of this rule and by label $(r, \texttt{o@}l)?x$ we have that $\Gamma'' = \mathit{sideEffect}((r, \texttt{o@}l)?x, \Gamma) = \Gamma$ by the definition of *sideEffect* (3.12). Therefore we are going to prove that $\Gamma \vdash_{\mathrm{B}} B' \triangleright \Gamma$. The thesis follows from applying rule T-Nil on $\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma$.

Case B-Choice

In this case, we know $B = \sum_{i \in J} \texttt{[}\eta_i\texttt{]\{}B_i\texttt{\}}$ for a sum of behaviours η_i and B_i over i:

$$(\texttt{B-Choice}) \quad \frac{j \in J \quad \eta_j \xrightarrow{\mu} Q_j}{\sum_{i \in J} \texttt{[}\eta_i\texttt{]\{}B_i\texttt{\}} \xrightarrow{\mu} Q_j ; B_j}$$

We also know that $B' = Q_j ; B_j$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{B}} \sum_{i \in J} \texttt{[}\eta_i\texttt{]\{}B_i\texttt{\}} \triangleright \Gamma'$ we know that it can only be typed by rule T-Choice:

$$(\texttt{T-Choice}) \quad \frac{\forall j \in J \ . \ \Gamma \vdash_{\mathrm{B}} \eta_j ; B_j \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \sum_{i \in J} \texttt{[}\eta_i\texttt{]\{}B_i\texttt{\}} \triangleright \Gamma'}$$

By applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{B}} \eta_j ; B_j \triangleright \Gamma'$ from this rule we know that it is only typable by T-Seq:

$$(\texttt{T-Seq}) \quad \frac{\Gamma \vdash_{\mathrm{B}} \eta_j \triangleright \Gamma''' \quad \Gamma''' \vdash_{\mathrm{B}} B_j \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \eta_j ; B_j \triangleright \Gamma'} \tag{3.19}$$

By applying the induction hypothesis on premise $\Gamma \vdash_{\mathrm{B}} \eta_j \triangleright \Gamma'''$ from 3.19 and premise $\eta_j \xrightarrow{\mu} Q_j$ from B-Choice we know $\Gamma'''' \vdash_{\mathrm{B}} Q_j \triangleright \Gamma'''$ where $\Gamma'''' = \mathit{sideEffect}(\mu, \Gamma)$ by the definition of *sideEffect* (3.12). Since the premise and the conclusion of B-Choice shares the same label we have $\Gamma'' = \Gamma''''$.

The thesis follows from applying T-Seq on $\Gamma'' \vdash_{\mathrm{B}} Q_j ; B_j \triangleright \Gamma'$ by $\Gamma'' \vdash_{\mathrm{B}} Q_j \triangleright \Gamma'''$ from the induction hypothesis and premise $\Gamma''' \vdash_{\mathrm{B}} B_j \triangleright \Gamma'$ from 3.19.

Case B-Struct

From the form of this rule we know that $B = B_1$:

$$(\texttt{B-Struct}) \quad \frac{B_1 \equiv B_2 \quad B_2 \xrightarrow{\mu} B_2' \quad B_1' \equiv B_2'}{B_1 \xrightarrow{\mu} B_1'}$$

We also know that $B' = B_1'$. By applying lemma 3.10 on $\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma'$ and premise $B_1 \equiv B_2$ we have $\Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'$. Applying the induction hypothesis on $\Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'$ and premise $B_2 \xrightarrow{\mu} B_2'$ we know $\Gamma'' \vdash_{\mathrm{B}} B_2' \triangleright \Gamma'$ where $\Gamma'' = \mathit{sideEffect}(\mu, \Gamma)$. Since the congruence relation is commutative we know $B_2' \equiv B_1'$ from premise $B_1' \equiv B_2'$. The thesis follows by applying lemma 3.10 on $\Gamma'' \vdash_{\mathrm{B}} B_2' \triangleright \Gamma'$ and premise $B_2' \equiv B_1'$ where $\Gamma'' = \mathit{sideEffect}(\mu, \Gamma)$.

□

3.3.4.2 Type Preservation at the Service Layer

Type preservation for processes are presented below followed by type preservation for services.

Type Preservation for Processes
Consider a well-typed process P typed with respect to an environment Γ. Assume there exists a process P' such that $P \xrightarrow{\mu} P'$. Then P' is also well typed with respect to Γ:

Theorem 3.16 (Type Preservation for Processes)

$$\begin{aligned} &\textit{If } \Gamma \vdash_{\mathrm{P}} P \\ &\textit{and } P \xrightarrow{\mu} P' \\ &\textit{then } \Gamma \vdash_{\mathrm{P}} P' \end{aligned}$$

Proof. Assume $\Gamma \vdash_{\mathrm{P}} P$ and $P \xrightarrow{\mu} P'$. The proof is done by induction on the derivation of $P \xrightarrow{\mu} P'$.

Case S-Read

In this case, we know $P = B \cdot t \cdot \tilde{m}$ for a process consisting of a behaviour B, a state t and a message queue $\tilde{m}$:

$$(\texttt{S-Read}) \quad \frac{B \xrightarrow{\mathbf{read}\ t} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\tau} B' \cdot t \cdot \tilde{m}}$$

We also know that $\mu = \tau$ and $P' = B' \cdot t \cdot \tilde{m}$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot \tilde{m}$ we know that it can only be typed using rule T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma'' \quad \Gamma,\Gamma' \vdash_{\mathrm{state}} t \quad \Gamma,\Gamma' \vdash_{\mathrm{queue}} \tilde{m} \quad \nexists o.\ o@l{:}\langle O\rangle \in \Gamma' \vee o{:}\langle O\rangle \in \Gamma'}{\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot \tilde{m}} \tag{3.20}$$

Applying theorem 3.13 on premise $\Gamma, \Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma''$ from 3.20 and premise $B \xrightarrow{\mathsf{read}\ t} B'$ from S-Read we know $\Gamma''' \vdash_{\mathrm{B}} B' \triangleright \Gamma''$ where $\Gamma''' = \mathit{sideEffect}(\mathsf{read}\ t, \Gamma, \Gamma') = \Gamma, \Gamma'$ by the definition of *sideEffect* (3.12). Since we know $t' = t$ and $\tilde{m}' = \tilde{m}$ by the form of S-Read, the thesis follows from applying T-Process on $\Gamma \vdash_{\mathrm{P}} B' \cdot t \cdot \tilde{m}$ by $\Gamma, \Gamma' \vdash_{\mathrm{B}} B' \triangleright \Gamma''$ and premise $\Gamma, \Gamma' \vdash_{\mathrm{state}} t$, $\Gamma, \Gamma' \vdash_{\mathrm{queue}} \tilde{m}$ and $\nexists \mathtt{o}.\, \mathtt{o@}l : \mathord{<}O\mathord{>} \in \Gamma' \vee \mathtt{o} : \mathord{<}O\mathord{>} \in \Gamma'$ from 3.20.

Case S-Send

The proof is similar to the proof for case S-Read.

$$(\texttt{S-Send}) \quad \frac{B \xrightarrow{\nu r\ \mathtt{o@}l(\mathtt{e})} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\nu r\ \mathtt{o@}l(\mathtt{e}(t))} B' \cdot t \cdot \tilde{m}}$$

Case S-Exec

The proof is similar to the proof for case S-Read.

$$(\texttt{S-Exec}) \quad \frac{B \xrightarrow{(r,\mathtt{o})!\, x} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{(r,\mathtt{o})!\, x(t)} B' \cdot t \cdot \tilde{m}}$$

Case S-Get

In this case, we know $P = B \cdot t \cdot (r, \mathtt{o}, t') :: \tilde{m}$ for a process consisting of a behaviour B, a state t and a message queue $(r, \mathtt{o}, t') :: \tilde{m}$:

$$(\texttt{S-Get}) \quad \frac{B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B'}{B \cdot t \cdot (r,\mathtt{o},t') :: \tilde{m} \xrightarrow{\tau} B' \cdot t \leftarrow_{\mathtt{x}} t' \cdot \tilde{m}}$$

We also know that $\mu = \tau$ and $P' = B' \cdot t \leftarrow_{\mathtt{x}} t' \cdot \tilde{m}$.

By applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot (r, \mathtt{o}, t') :: \tilde{m}$ we know that it can only be typed by rule T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma, \Gamma'' \vdash_{\mathrm{B}} B \triangleright \Gamma''' \quad \Gamma, \Gamma'' \vdash_{\mathrm{state}} t \quad \Gamma, \Gamma'' \vdash_{\mathrm{queue}} (r,\mathtt{o},t') :: \tilde{m} \quad \nexists \mathtt{o}.\, \mathtt{o@}l : \mathord{<}O\mathord{>} \in \Gamma' \vee \mathtt{o} : \mathord{<}O\mathord{>} \in \Gamma''}{\Gamma \vdash_{\mathrm{P}} B \cdot t \cdot (r,\mathtt{o},t') :: \tilde{m}} \tag{3.21}$$

Let $\Gamma' = \Gamma, \Gamma''$. By applying the Preservation for Behavioural Layer Theorem (theorem 3.13) on premise $\Gamma, \Gamma'' \vdash_{\mathrm{B}} B \triangleright \Gamma'''$ from 3.21 and on premise $B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B'$ from S-Get we get:

$$\begin{aligned}&\text{If } \Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma''' \\ &\text{and } B \xrightarrow{r:\mathtt{o(x)}} B' \\ &\text{then } \Gamma'''' \vdash B' \triangleright \Gamma''' \\ &\text{where } \Gamma'''' = \mathit{sideEffect}(r : \mathtt{o(x)}, \Gamma')\end{aligned} \tag{3.22}$$

From the definition of *sideEffect* (definition 3.12) we know that there exist two cases for Γ'''':

Subcase $\Gamma'''' = \Gamma'$

Lemma 3.17 *If* $\Gamma \vdash_{\text{queue}} (r, \mathtt{o}, t') :: \tilde{m}$ *then* $\Gamma \vdash_{\text{queue}} \tilde{m}$.

Proof. The proof follows immediately from definition 3.8. □

By $\Gamma' \vdash_{\text{queue}} (r, \mathtt{o}, t') :: \tilde{m}$ from 3.21 and by lemma 3.17 we have:

$$\Gamma' \vdash_{\text{queue}} \tilde{m} \tag{3.23}$$

and we know by definition 3.8 that data tree t' fulfills the requirements for operation $\mathtt{o}$ defined in environment Γ':

$$(\mathtt{o} : \texttt{<}T_i\texttt{>} \in \Gamma' \vee \mathtt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma') \wedge \vdash t' : T_{t'} \wedge T_{t'} \leq T_i \tag{3.24}$$

From the definition of *sideEffect* (definition 3.12) we know that the input through operation o is going to be stored in an existing variable, x, which type is a super type of the declared input type of o:

$$x : T_x \in \Gamma' \wedge (\mathtt{o} : \texttt{<}T_i\texttt{>} \in \Gamma' \vee \mathtt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma') \wedge T_i \leq T_x \tag{3.25}$$

Lemma 3.18 *If* $x : T_x \in \Gamma'$ *then* $r(x) \in \mathit{Roots}(\Gamma')$

Proof. The proof follows from the form of the typing system. □

By lemma 3.18 applied on $x : T_x \in \Gamma'$ from 3.25 we get $r(x) \in \mathit{Roots}(\Gamma')$.
Since we have $\Gamma' \vdash_{\text{state}} t$ from 3.21 we know by definition 3.7 that

$$r(x) : T_{r(x)} \in \Gamma' \wedge \vdash (r(x))(t) : T' \wedge T' \leq T_{r(x)} \tag{3.26}$$

By 3.24 and 3.25 we know $T_{t'} \leq T_i \leq T_x$.
Since $T_{t'} \leq T_x$ and $T' \leq T_{r(x)}$ and $r(x) \in \mathit{Roots}(\Gamma')$ then by $\Gamma' \vdash_{\text{state}} t$ from 3.21 we know by definition 3.7 that

$$\forall a : T_a \in \mathit{Roots}(\Gamma'). \vdash (a)(t \leftarrow_{\mathtt{x}} t') : T'' \wedge T'' \leq T_a \tag{3.27}$$

Since $x \in \mathit{Roots}(\Gamma')$ we know by $\Gamma' \vdash_{\text{state}} t$ from 3.21 that

$$\forall a \in \mathit{Roots}(t). a \in \mathit{Roots}(\Gamma') \tag{3.28}$$

by definition 3.7.
From 3.27 and 3.28 we have

$$\Gamma' \vdash_{\text{state}} t \leftarrow_{\mathtt{x}} t' \tag{3.29}$$

The thesis follows from applying T-Process on $\Gamma' \vdash B' \triangleright \Gamma''$ from 3.22 and on 3.23 and 3.29.

Subcase $\Gamma'''' = \mathit{upd}(\Gamma', x, T_i)$

Since no variable type declaration is used from the environment when typing a message queue with respect to an environment then by $\Gamma' \vdash_{\text{queue}} (r, \mathtt{o}, t') :: \tilde{m}$ from 3.21 and by lemma 3.17 we know:

$$\Gamma'''' \vdash_{\text{queue}} \tilde{m} \tag{3.30}$$

We also know that data tree t' fulfills the requirements for operation $\mathtt{o}$ defined in environment Γ' as shown in 3.24.
From the definition of *sideEffect* (definition 3.12) we know ($\mathtt{o} : \texttt{<}T_i\texttt{>} \in \Gamma'''' \vee \mathtt{o} : \texttt{<}T_i, T_o\texttt{>} \in \Gamma''''$). The environment is updated with a new path x, which gets the same type as the declared type for the input part of o. If part of the path to the leaf of x is missing, it is created with type `void` as basic type for each missing step.
By the definition of *sideEffect* and 3.24 we know

$$t' : T_{t'} \wedge \Gamma'''' \vdash x : T_i \wedge T_{t'} \leq T_i \tag{3.31}$$

By the form of Γ'''' and lemma 3.18 we know $r(x) \in \mathit{Roots}(\Gamma'''')$. From $r(x) \in \mathit{Roots}(\Gamma'''')$, 3.31, the definition of *upd* and $\Gamma' \vdash_{\text{state}} t$ from 3.21 we know:

$$\forall a : T_a \in \mathit{Roots}(\Gamma''''). \vdash (a)(t \leftarrow_{\mathtt{x}} t') : T \wedge T \leq T_a \tag{3.32}$$

The difference between Γ' and Γ'''' as well as t and $t \leftarrow_{\mathtt{x}} t'$ is that path x and its subnodes are added. Therefore we have by $\Gamma' \vdash_{\text{state}} t$ from 3.21 that

$$\forall a \in \mathit{Roots}(t \leftarrow_{\mathtt{x}} t'). a \in \mathit{Roots}(\Gamma'''') \tag{3.33}$$

From 3.32 and 3.33 we know $\Gamma'''' \vdash_{\text{state}} t \leftarrow_{\mathtt{x}} t'$. The thesis follows from applying T-Process on $\Gamma'''' \vdash B' \triangleright \Gamma''$ from 3.22 and on $\Gamma'''' \vdash_{\text{state}} t \leftarrow_{\mathtt{x}} t'$ and 3.30.

Case S-Assign

In this case, we know $P = B \cdot t \cdot \tilde{m}$ for a process consisting of a behaviour B, a state t and a message queue $\tilde{m}$:

$$(\texttt{S-Assign}) \quad \frac{B \xrightarrow{\mathtt{x}=e} B'}{B \cdot t \cdot \tilde{m} \xrightarrow{\tau} B' \cdot t \twoheadleftarrow_{\mathtt{x}} e(t) \cdot \tilde{m}}$$

We also know that $\mu = \tau$ and $P' = B' \cdot t \twoheadleftarrow_{\mathtt{x}} e(t) \cdot \tilde{m}$.

By applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\text{P}} B \cdot t \cdot \tilde{m}$ we know that it can only be typed by rule T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma' \vdash_{\text{B}} B \triangleright \Gamma''' \quad \Gamma,\Gamma' \vdash_{\text{state}} t \quad \Gamma,\Gamma' \vdash_{\text{queue}} \tilde{m} \quad \nexists \mathtt{o}.\, \mathtt{o}@l{:}\langle O\rangle \in \Gamma' \vee \mathtt{o}{:}\langle O\rangle \in \Gamma'}{\Gamma \vdash_{\text{P}} B \cdot t \cdot \tilde{m}} \tag{3.34}$$

Let $\Gamma' = \Gamma, \Gamma''$. By applying the Preservation for Behavioural Layer Theorem (theorem 3.13) on premise $\Gamma, \Gamma' \vdash_{\text{B}} B \triangleright \Gamma'''$ from 3.34 and on premise $B \xrightarrow{\mathtt{x}=e} B'$ from S-Assign we get:

$$\begin{aligned} &\text{If } \Gamma' \vdash_{\text{B}} B \triangleright \Gamma''' \\ &\text{and } B \xrightarrow{\mathtt{x}=e} B' \\ &\text{then } \Gamma'''' \vdash B' \triangleright \Gamma''' \\ &\text{where } \Gamma'''' = \mathit{sideEffect}(\mathtt{x}=e, \Gamma') \end{aligned} \tag{3.35}$$

From the definition of *sideEffect* (definition 3.12) we know that there exist two cases for Γ'''':

Subcase $\Gamma'''' = \Gamma'$

From 3.34 we have $\Gamma' \vdash_{\text{queue}} \tilde{m}$. Since $\Gamma'''' = \Gamma'$ we must also have

$$\Gamma'''' \vdash_{\text{queue}} \tilde{m} \tag{3.36}$$

From the definition of *sideEffect* (definition 3.12) we know that x already exists in Γ' and that its basic type equals the basic type of expression e:

$$\Gamma' \vdash e : T_e \wedge x : T_x \in \Gamma' \wedge bt(T_e) = bt(T_x) \tag{3.37}$$

We therefore have that

$$T = T' \text{ where } \vdash r(x)(t) : T \ \wedge \ (r(x))(t \twoheadleftarrow_{\mathtt{x}} e(t)) : T' \tag{3.38}$$

Since x is an existing path and by 3.38 and by $\Gamma' \vdash_{\text{state}} t$ from 3.34 we know by definition 3.7 that

$$\Gamma' \vdash_{\text{state}} t \twoheadleftarrow_{\mathtt{x}} e(t) \tag{3.39}$$

The thesis follows from applying T-Process on $\Gamma' \vdash B' \triangleright \Gamma'''$ from 3.35 and on 3.36 and 3.39.

Subcase $\Gamma'''' = upd(\Gamma', x, bt(T_x))$

By definition 3.8 we know that when typing a message queue with respect to an environment no variable type declaration is used from the environment. We therefore have

$$\Gamma'''' \vdash_{\text{queue}} \tilde{m} \tag{3.40}$$

From the definition of *sideEffect* (definition 3.12) we know $\Gamma'''' \vdash x : bt(T_e)$, where $\Gamma' \vdash e : T_e \wedge x \notin \Gamma'$. The environment is updated with a new path x which gets the basic type of the result of the evaluation of expression e. If part of the path to the leaf of x is missing, it is created with type `void` as basic type for each missing step.
By the form of Γ'''' and by lemma 3.18 we know $r(x) \in Roots(\Gamma'''')$. From $r(x) \in Roots(\Gamma'''')$, the definition of *sideEffect*, the definition of *upd* (definition 3.6) and $\Gamma' \vdash_{\text{state}} t$ from 3.34 we have:

$$\forall a : T_a \in \mathit{Roots}(\Gamma''''). \vdash a(t \twoheadleftarrow_{\mathtt{x}} e(t)) : T \wedge T \leq T_a \tag{3.41}$$

The difference between Γ' and Γ'''' as well as t and $t \twoheadleftarrow_{\mathtt{x}} e(t)$ is that path x and all its subnodes are added. Therefore we have by $\Gamma' \vdash_{\text{state}} t$ from 3.34 that

$$\forall a \in \mathit{Roots}(t \twoheadleftarrow_{\mathtt{x}} e(t)). a \in \mathit{Roots}(\Gamma'''') \tag{3.42}$$

From 3.41 and 3.42 we know $\Gamma'''' \vdash_{\text{state}} t \twoheadleftarrow_{\mathtt{x}} e(t)$. The thesis follows from applying T-Process on $\Gamma' \vdash B' \triangleright \Gamma'''$ from 3.35 and on $\Gamma'''' \vdash_{\text{state}} t \twoheadleftarrow_{\mathtt{x}} e(t)$ and 3.40.

Case S-Wait

The proof is similar to the proof for case S-Get.

Case S-Par

The proof is straightforward because the typing of operations in an environment does not change.

□

Type Preservation for Services
Consider a well-typed service S typed with respect to an environment Γ. Assume there exists a service S' such that $S \xrightarrow{\mu} S'$. Then S' is also well typed with respect to Γ:

Theorem 3.19 (Type Preservation for Services)

$$\begin{aligned} &\textit{If } \Gamma \vdash_{\mathrm{S}} S \\ &\textit{and } S \xrightarrow{\mu} S' \\ &\textit{then } \Gamma \vdash_{\mathrm{S}} S' \end{aligned}$$

Proof. Assume $\Gamma \vdash_{\mathrm{S}} S$ and $S \xrightarrow{\mu} S'$. The proof is done by induction on the derivation of $S \xrightarrow{\mu} S'$.

Case S-Corr

In this case, we know $S = B_1 \triangleright_D P \mid B_2 \cdot t \cdot \tilde{m}$ for a behaviour B_1, a deployment part D and a process consisting of a process P and another process with the behaviour B_2, the state t and the message queue $\tilde{m}$:

$$(\texttt{S-Corr}) \quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad t',\mathtt{o} \vdash_{\alpha_C} t \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'} \leq T_i}{B_1 \triangleright_D P \mid B_2{\cdot}t{\cdot}\tilde{m} \xrightarrow{\nu r\ \mathtt{o}(t')} B_1 \triangleright_D P \mid B_2{\cdot}t{\cdot}\tilde{m}{::}(r,\mathtt{o},t')}$$

We also know that $\mu = \boldsymbol{\nu} r\ \mathtt{o}(t')$ and $S' = B_1 \triangleright_D P \mid B_2 \cdot t \cdot \tilde{m} :: (r, \mathtt{o}, t')$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{S}} B_1 \triangleright_D P \mid B_2 \cdot t \cdot \tilde{m}$ we know that it can only be typed by rule T-Service:

$$(\texttt{T-Service}) \quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B_1 \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P \mid B_2{\cdot}t{\cdot}\tilde{m}}{\Gamma \vdash_{\mathrm{S}} B_1 \triangleright_D P \mid B_2{\cdot}t{\cdot}\tilde{m}} \tag{3.43}$$

Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{P}} P \mid B_2 \cdot t \cdot \tilde{m}$ we know that it can only be typed by rule T-Process-Par:

$$(\texttt{T-Process-Par}) \quad \frac{\Gamma \vdash_{\mathrm{P}} P \quad \Gamma \vdash_{\mathrm{P}} B_2{\cdot}t{\cdot}\tilde{m}}{\Gamma \vdash_{\mathrm{P}} P \mid B_2{\cdot}t{\cdot}\tilde{m}} \tag{3.44}$$

Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{P}} B_2 \cdot t \cdot \tilde{m}$ from this rule, we know that it can only be typed by rule T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma''' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma' \quad \Gamma,\Gamma''' \vdash_{\mathrm{state}} t \quad \Gamma,\Gamma''' \vdash_{\mathrm{queue}} \tilde{m} \quad \nexists \mathtt{o}.\, \mathtt{o@}l{:}\langle O\rangle{\in}\Gamma''' \vee \mathtt{o}{:}\langle O\rangle{\in}\Gamma'''}{\Gamma \vdash_{\mathrm{P}} B_2{\cdot}t{\cdot}\tilde{m}} \tag{3.45}$$

Let $\Gamma'' = \Gamma, \Gamma'''$. By premise $\Gamma'' \vdash_{\mathrm{queue}} \tilde{m}$ from 3.45 and by premise $(\mathtt{o} : \langle T_i\rangle \in \Gamma \vee \mathtt{o} : \langle T_i, T_o\rangle \in \Gamma)$, $\vdash t' : T_{t'}$ and $T_{t'} \leq T_i$ from S-Corr we know that $\Gamma'' \vdash_{\mathrm{queue}} \tilde{m} :: (r, \mathtt{o}, t')$ according to definition 3.8. By $\Gamma'' \vdash_{\mathrm{queue}} \tilde{m} :: (r, \mathtt{o}, t')$ and premise $\Gamma'' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'$, $\Gamma'' \vdash_{\mathrm{state}} t$ and $\nexists \mathtt{o}.\, \mathtt{o@}l : \langle O\rangle \in \Gamma''' \vee \mathtt{o} : \langle O\rangle \in \Gamma'''$ from 3.45 we can apply T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma''' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma' \quad \Gamma,\Gamma''' \vdash_{\mathrm{state}} t \quad \Gamma,\Gamma''' \vdash_{\mathrm{queue}} \tilde{m}{::}(r,\mathtt{o},t') \quad \nexists \mathtt{o}.\, \mathtt{o@}l{:}\langle O\rangle{\in}\Gamma''' \vee \mathtt{o}{:}\langle O\rangle{\in}\Gamma'''}{\Gamma \vdash_{\mathrm{P}} B_2{\cdot}t{\cdot}\tilde{m}{::}(r,\mathtt{o},t')} \tag{3.46}$$

By applying T-Process-Par on premise $\Gamma \vdash_{\mathrm{P}} P$ from 3.44 and on $\Gamma \vdash_{\mathrm{P}} B_2 \cdot t \cdot \tilde{m} :: (r, \mathtt{o}, t')$ from 3.46 we get:

$$(\texttt{T-Process-Par}) \quad \frac{\Gamma \vdash_{\mathrm{P}} P \quad \Gamma \vdash_{\mathrm{P}} B_2{\cdot}t{\cdot}\tilde{m}{::}(r,\mathtt{o},t')}{\Gamma \vdash_{\mathrm{P}} P \mid B_2{\cdot}t{\cdot}\tilde{m}{::}(r,\mathtt{o},t')} \tag{3.47}$$

The thesis follows from applying T-Service on $\Gamma \vdash_{\mathrm{P}} P \mid B_2 \cdot t \cdot \tilde{m} :: (r, \mathtt{o}, t')$ from 3.47 and premise $D = \alpha_C \cdot \Gamma$ and $\Gamma \vdash_{\mathrm{BSL}} B_1 \triangleright \Gamma'$ from 3.43.

Case S-Start

In this case, we know $S = B \triangleright_D P$ for a behaviour B, a deployment part D and a process P:

$$\text{(S-Start)}\quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad t{,}\mathtt{o} \not\leftarrow_{\alpha_C} P \qquad B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B' \qquad t'{=}\mathsf{init}(t,\mathtt{o},\alpha_C) \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'}{\leq}T_i}{B \triangleright_D P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}(t)} B \triangleright_D P \mid B'{\cdot}t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t'{\cdot}\epsilon}$$

We also know that $\mu = \boldsymbol{\nu} r\,\mathtt{o}(t)$ and $S' = B \triangleright_D P \mid B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ we know that it can only be typed by rule T-Service:

$$\text{(T-Service)}\quad \frac{D{=}\alpha_C{\cdot}\Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P} \tag{3.48}$$

Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma'$ we know that it is typed either with T-BSL-Choice or T-BSL-Nil. Since a nil behaviour can not take a step of evaluation according to the semantics B must be a choice behaviour. Therefore can B only be typed using T-BSL-Choice.

$$\text{(T-BSL-Choice)}\quad \frac{\forall j{\in}J \;.\; \Gamma \vdash_{\mathrm{B}} \eta_j; B_j \triangleright \Gamma_j \quad \nexists \mathrm{T}_k,\mathrm{T}_l \;.\; x{:}\mathrm{T}_k \in \bigcup \Gamma_j \wedge x{:}\mathrm{T}_l \in \bigcup \Gamma_j \wedge \mathrm{T}_k{\neq}\mathrm{T}_l}{\Gamma \vdash_{\mathrm{BSL}} \sum_{i\in J} [\eta_i]\{B_i\} \triangleright \bigcup \Gamma_{j\in J}}$$

Applying theorem 3.13 on premise $\Gamma \vdash_{\mathrm{B}} \eta_j; B_j \triangleright \Gamma_j$ from this rule and premise $B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B'$ from S-Start we get $\Gamma'' \vdash_{\mathrm{B}} B' \triangleright \Gamma'$ where $\Gamma'' = \mathit{sideEffect}(\mathtt{x} = e, \Gamma)$. Since Γ only contains operation type information we know $\Gamma'' = \mathit{upd}(\Gamma, x, T_i)$ by the definition of *sideEffect* (3.12).

By Γ'' and premise $(\mathtt{o} : \langle T_i\rangle \in \Gamma \vee \mathtt{o} : \langle T_i, T_o\rangle \in \Gamma), \vdash t' : T_{t'}$ and $T_{t'} \leq T_i$ from S-Start we have $\Gamma'' \vdash_{\mathrm{state}} t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t'$ according to definition 3.7. By Γ'' we have $\Gamma'' \vdash_{\mathrm{queue}} \epsilon$ according to definition 3.8.

Let $\Gamma, \Gamma''' = \Gamma''$.By applying $\Gamma, \Gamma''' \vdash_{\mathrm{B}} B' \triangleright \Gamma'$, $\Gamma, \Gamma''' \vdash_{\mathrm{state}} t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t'$ and $\Gamma, \Gamma''' \vdash_{\mathrm{queue}} \epsilon$ on T-Process we get:

$$\text{(T-Process)}\quad \frac{\Gamma,\Gamma''' \vdash_{\mathrm{B}} B' \triangleright \Gamma' \quad \Gamma,\Gamma''' \vdash_{\mathrm{state}} t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \quad \Gamma,\Gamma''' \vdash_{\mathrm{queue}} \epsilon \quad \nexists \mathtt{o}.\ \mathtt{o}@l{:}\langle O\rangle \in \Gamma''' \vee \mathtt{o}{:}\langle O\rangle \in \Gamma'''}{\Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon} \tag{3.49}$$

By applying on T-process-par premise $\Gamma \vdash_{\mathrm{P}} P$ from 3.48 and $\Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon$ from 3.49 we get:

$$\text{(T-Process-Par)}\quad \frac{\Gamma \vdash_{\mathrm{P}} P \quad \Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon}{\Gamma \vdash_{\mathrm{P}} P \mid \Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon} \tag{3.50}$$

The thesis follows from applying T-Service on $B \triangleright_D P \mid B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon$ by conclusion $\Gamma \vdash_{\mathrm{P}} P \mid \Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon$ from 3.50 and premise $D = \alpha_C \cdot \Gamma$ and $\Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma'$ from 3.48:

$$(\mathtt{T\text{-}Service}) \quad \frac{D = \alpha_C \cdot \Gamma \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P \mid \Gamma \vdash_{\mathrm{P}} B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P \mid B' \cdot t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon}$$

Case S-Send-Lift

If the last rule in the derivation sequence is S-Send-Lift, then from the form of this rule, we see that $S = B \triangleright_D P$ for a behaviour B, a deployment part D and a process P:

$$(\mathtt{S\text{-}Send\text{-}Lift}) \quad \frac{P \xrightarrow{\boldsymbol{\nu} r\, \mathtt{o@}l(\mathtt{t})} P'}{B \triangleright_D P \xrightarrow{\boldsymbol{\nu} r\, \mathtt{o@}l(\mathtt{t})} B \triangleright_D P'}$$

We also know that $\mu = \boldsymbol{\nu} r\ \mathtt{o@}l(\mathtt{t})$ and $S' = B \triangleright_D P'$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ we know that it can only be typed by rule T-Service:

$$(\mathtt{T\text{-}Service}) \quad \frac{D = \alpha_C \cdot \Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P} \tag{3.51}$$

Applying theorem 3.16 on premise $\Gamma \vdash_{\mathrm{P}} P$ from 3.51 and premise $P \xrightarrow{\boldsymbol{\nu} r\, \mathtt{o@}l(\mathtt{t})} P'$ from S-Send-Lift we get $\Gamma \vdash_{\mathrm{P}} P'$. The thesis follows from applying T-Service on $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P'$ by $\Gamma \vdash_{\mathrm{P}} P'$ and premise $D = \alpha_C \cdot \Gamma$ and $\Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma'$ from 3.51.

Case S-Exec-Lift

The proof is similar to the proof for case S-Send-Lift.

Case S-Wait-Lift

The proof is similar to the proof for case S-Send-Lift.

Case S-Tau

The proof is similar to the proof for case S-Send-Lift.

□

3.3.4.3 Type Preservation at the Network Layer

Consider a well-typed network N typed with respect to an environment Γ. Assume there exists a network N' such that $N \xrightarrow{\mu} N'$. Then N' is also well typed with respect to Γ:

Theorem 3.20 (Type Preservation for Networks)

$$\begin{array}{l} \textit{If } \Gamma \vdash_{\mathrm{N}} N \\ \textit{and } N \xrightarrow{\mu} N' \\ \textit{then } \Gamma \vdash_{\mathrm{N}} N' \end{array}$$

Proof. Assume $\Gamma \vdash_{\mathrm{N}} N$ and $N \xrightarrow{\mu} N'$. The proof is done by induction on the derivation of $N \xrightarrow{\mu} N'$.

Case N-Comm

In this case, we know $N = [S_1]_{l_1} \mid [S_2]_{l_2}$ for service S_1 and S_2 with locations respectively l_1 and l_2:

$$(\texttt{N-Comm}) \quad \frac{S_1 \xrightarrow{\boldsymbol{\nu} r\, \mathtt{o}@l_2(t_M)} S_1' \quad S_2 \xrightarrow{\boldsymbol{\nu} r\, \mathtt{o}(t_M)} S_2' \quad r \notin \mathtt{cn}(S_1) \cup \mathtt{cn}(S_2)}{[S_1]_{l_1} \mid [S_2]_{l_2} \xrightarrow{\tau} \boldsymbol{\nu} r\ (\ [S_1']_{l_1} \mid [S_2']_{l_2}\ )}$$

We also know that $\mu = \tau$ and $N' = \boldsymbol{\nu} r\ (\ [S_1']_{l_1} \mid [S_2']_{l_2}\ )$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{N}} [S_1]_{l_1} \mid [S_2]_{l_2}$ we know that it can only be typed using T-Network:

$$(\texttt{T-Network}) \quad \frac{\begin{array}{c} \Gamma_1 \vdash_{\mathrm{N}} [S_1]_{l_1} \quad \Gamma_2 \vdash_{\mathrm{N}} [S_2]_{l_2} \\ \forall \mathtt{o}@l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\ \mathtt{o}@l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \mathtt{o}@l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\ \mathtt{o}@l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\mathtt{o}@l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \mathtt{o}@l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2) \end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S_1]_{l_1} \mid [S_2]_{l_2}} \tag{3.52}$$

where $\Gamma_1 \cup \Gamma_2 = \Gamma$. Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma_1 \vdash_{\mathrm{N}} [S_1]_{l_1}$ from this rule we know that it can only be typed using T-Deployment:

$$(\texttt{T-Deployment}) \quad \frac{\Gamma' \vdash_{\mathrm{S}} S_1 \quad l_1 \notin locs(\Gamma')}{\{\mathtt{o}@l : \texttt{<}O\texttt{>} \in \Gamma'\} \cup \{\mathtt{o}@l_1 : \texttt{<}O\texttt{>} \mid \mathtt{o} : \texttt{<}O\texttt{>} \in \Gamma'\} \vdash_{\mathrm{N}} [S_1]_{l_1}} \tag{3.53}$$

where $\Gamma_1 = \{\texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma\} \cup \{\texttt{o@}l_1 : \texttt{<}O\texttt{>} | \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma\}$. By applying theorem 3.19 at premise $\Gamma' \vdash_{\mathrm{S}} S_1$ from 3.53 and premise $S_1 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l_2(t_M)} S_1'$ from N-Comm we get $\Gamma' \vdash_{\mathrm{S}} S_1'$. Applying T-Deployment on $\Gamma' \vdash_{\mathrm{S}} S_1'$ and premise $l_1 \notin locs(\Gamma')$ from 3.53 we know $\Gamma_1 \vdash_{\mathrm{N}} [S_1']_{l_1}$. By similar argumentation we know $\Gamma_2 \vdash_{\mathrm{N}} [S_2']_{l_2}$. Applying $\Gamma_1 \vdash_{\mathrm{N}} [S_1']_{l_1}$, $\Gamma_2 \vdash_{\mathrm{N}} [S_2']_{l_2}$ and premise $\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1$ where $l \in locs(N_2).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1)$, $\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2$ where $l \in locs(N_1).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2)$ and $\neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)$ from 3.52 we get:

$$\text{(T-Network)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} [S_1']_{l_1} \quad \Gamma_2 \vdash_{\mathrm{N}} [S_2']_{l_2} \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S_1']_{l_1} \mid [S_2']_{l_2}} \tag{3.54}$$

where $\Gamma_1 \cup \Gamma_2 = \Gamma$. By $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S_1']_{l_1} \mid [S_2']_{l_2}$ we can apply T-Restriction:

$$\text{(T-Restriction)}\quad \frac{\Gamma \vdash_{\mathrm{N}} [S_1']_{l_1} \mid [S_2']_{l_2}}{\Gamma \vdash_{\mathrm{N}} \boldsymbol{\nu} r\ (\ [S_1']_{l_1} \mid [S_2']_{l_2}\ )}$$

The thesis follows from the conclusion of this rule.

Case N-Tau

In this case, we know $N = [S]_l \mid N$ for a service S, a location l and network N:

$$\text{(N-Tau)}\quad \frac{S \xrightarrow{\tau} S'}{[S]_l \mid N \xrightarrow{\tau} [S']_l \mid N}$$

We also know that $\mu = \tau$ and $N' = [S']_l \mid N$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{N}} [S]_l \mid N$ we know that it can only be typed using T-Network:

$$\text{(T-Network)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} [S]_l \quad \Gamma_2 \vdash_{\mathrm{N}} N \\ \forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l' \in locs(N).\ \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l' \notin locs([S]_l) \\ \forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l' \in locs([S]_l).\ \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l' \notin locs(N) \\ \neg(\texttt{o@}l' : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l' : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S]_l \mid N} \tag{3.55}$$

where $\Gamma = \Gamma_1 \cup \Gamma_2$. Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma_1 \vdash_{\mathrm{N}} [S]_l$ from 3.55 we know that it can only be typed using T-Deployment:

$$(\texttt{T-Deployment}) \quad \frac{\Gamma' \vdash_{\mathrm{S}} S \quad l \notin locs(\Gamma')}{\{\texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma'\} \cup \{\texttt{o@}l : \texttt{<}O\texttt{>} | \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma'\} \vdash_{\mathrm{N}} [S]_l} \tag{3.56}$$

where $\Gamma_1 = \{\texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma'\} \cup \{\texttt{o@}l : \texttt{<}O\texttt{>} | \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma'\}$. Applying theorem 3.19 on premise $\Gamma' \vdash_{\mathrm{S}} S$ from 3.56 and premise $S \xrightarrow{\tau} S'$ from N-Tau we know $\Gamma' \vdash_{\mathrm{S}} S'$.

Lemma 3.21 *Operation type declarations in an environment* Γ *are never changed. Neither doing a transition nor doing the type checking.*

Proof. The proof follows from the definition of *sideEffect* and the form of the typing system. □

By $\Gamma' \vdash_{\mathrm{S}} S'$, lemma 3.21 and premise $l \notin locs(\Gamma')$ from 3.56 we can apply T-Deployment:

$$(\texttt{T-Deployment}) \quad \frac{\Gamma' \vdash_{\mathrm{S}} S' \quad l \notin locs(\Gamma')}{\{\texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma'\} \cup \{\texttt{o@}l : \texttt{<}O\texttt{>} | \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma'\} \vdash_{\mathrm{N}} [S']_l} \tag{3.57}$$

By lemma 3.21 and $\{\texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma'\} \cup \{\texttt{o@}l : \texttt{<}O\texttt{>} | \texttt{o} : \texttt{<}O\texttt{>} \in \Gamma'\} \vdash_{\mathrm{N}} [S']_l$ and premise $\Gamma_2 \vdash_{\mathrm{N}} N$, $\forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1$ where $l' \in locs(N).\, \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l' \notin locs([S]_l)$, $\forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2$ where $l' \in locs([S]_l).\, \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l' \notin locs(N)$ and $\neg(\texttt{o@}l' : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l' : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)$ from 3.55 we can apply N-Network on $\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S']_l \mid N$ where $\Gamma = \Gamma_1 \cup \Gamma_2$:

$$(\texttt{T-Network}) \quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} [S']_l \quad \Gamma_2 \vdash_{\mathrm{N}} N \\ \forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l' \in locs(N).\, \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l' \notin locs([S]_l) \\ \forall \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l' \in locs([S]_l).\, \texttt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l' \notin locs(N) \\ \neg(\texttt{o@}l' : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l' : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} [S]_l \mid N} \tag{3.58}$$

The thesis follows from the conclusion of this rule 3.58.

Case N-Response

The proof is similar to the proof for N-Comm, except that the rules are applied in a different order.

Case N-Struct

The proof is similar to the proof for case P-Struct.

Case N-Par

The proof is straightforward because the typing of operations in an environment does not change.

□

3.4 Type Safety

The purpose of the type system is to ensure that a message with the wrong type is never send nor received. We can formalize this to the type safety property:

1. A well-typed statement is also well typed after taking a transition.
2. A well-typed statement can not take a transition labeled `error`.

The part 1 is called preservation and is presented in 3.3.4. The part of the type safety property that makes it address the purpose of this type system lays in the formulation of semantic rules with label `error`. We formulate them to present situations that violates the purpose of the type system and we extend the semantics of Jolie with these rules in 3.4.1. Because the purpose of the type system is regarding communication the new extention only consists of rules at the service and network layers. In 3.4.2 we present 2 and in 3.4.3 we collect the parts and present type safety.

3.4.1 Semantics with Errors

We extend the semantics of Jolie presented in chapter 2 with rules for communiation performed by a network and messages send by a services in which there is type mismatch. For the receiving of a message the type system relies on the dynamic type check.

E-Comm Two services running in parallel can take a transition labelled `error` if they perform a unidirectional communication or the first part of a bidirectional communication in which there is a type mismatch between the services's type declarations for the operation used.

$$(\texttt{E-Comm}) \quad \frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad (\texttt{o@}l_2 : \texttt{<}T_1\texttt{>} \in \Gamma_1 \vee \texttt{o@}l_2 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1) \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad (\texttt{o} : \texttt{<}T_2\texttt{>} \in \Gamma_2 \vee \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2) \quad T_1 \neq T_2 \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l_2(t)} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o}(t)} B_2 \triangleright_{D_2} P_2' \quad r \notin \texttt{cn}(S_1) \cup \texttt{cn}(S_2) \end{array}}{[B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2} \xrightarrow{\texttt{error}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}}$$

E-Response Two services running in parallel can take a transition labelled `error` if they perform the second part of a bidirectional communication in which there is a type mismatch between the services's type declarations for the operation used.

$$(\texttt{E-Response}) \quad \frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad \texttt{o@}l_1 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1 \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2 \quad T_2' \neq T_1' \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{(r, \texttt{o@}l_1)?t} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{(r, \texttt{o})!\,t} B_2 \triangleright_{D_2} P_2' \end{array}}{\boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ ) \xrightarrow{\texttt{error}} \boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )}$$

E-Send-Lift A service can take a transition labelled `error` if it can send a message which type is not a subtype of the type declared for the first message communicated using the operation

$$(\texttt{E-Send-Lift}) \quad \frac{D = \alpha_C \cdot \Gamma \quad (\texttt{o@}l : \texttt{<}T_o\texttt{>} \in \Gamma \vee \texttt{o@}l : \texttt{<}T_o, T_i\texttt{>} \in \Gamma) \quad \vdash t : T_t \quad T_t \not\leq T_o \quad P \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l(\texttt{t})} P'}{B \triangleright_D P \xrightarrow{\texttt{error}} B \triangleright_D P}$$

E-Exec-Lift A service can take a transition labelled `error` if it can send a message which type is not a subtype of the type declared for the second message communicated using the operation.

$$(\texttt{E-Exec-Lift}) \quad \frac{D = \alpha_C \cdot \Gamma \quad \texttt{o@}l : \texttt{<}T_i, T_o\texttt{>} \in \Gamma \quad \vdash t : T_t \quad T_t \not\leq T_o \quad P \xrightarrow{(r, \texttt{o})!\,t} P'}{B \triangleright_D P \xrightarrow{\texttt{error}} B \triangleright_D P}$$

3.4.2 Lack of Errors

In order to ensure that a statement can not in an evaluation take a step labeled `error`, we must ensure that it can not take a single step labeled `error`. We do this in Lemma 3.22 and 3.23. presented below. The two Lemmas address

the comunication issue at respectively service layer and network layer, since the purpose of the type system is to ensure that a message with the wrong type is never send nor received.

The property that a well-typed service can not take a transition labeled `error` is described in the following Lemma.

Lemma 3.22 (Lack of Errors at Service Layer)

Let $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$
then $B \triangleright_D P \xrightarrow{\mu} B \triangleright_D P'$
implies $\mu \neq \texttt{error}$

Proof.

Assume $\Gamma \vdash_{\mathrm{S}} S$ and $S \xrightarrow{\mu} S'$. The proof is done by induction on the derivation of $S \xrightarrow{\mu} S'$.

We now consider the base cases:

Case E-Send-Lift

If the last rule in the derivation sequence is E-Send-Lift, then from the form of this rule, we see that $S = B \triangleright_D P$ for a behaviour B, a deployment part D and a process P:

$$(\texttt{E-Send-Lift}) \quad \frac{D=\alpha_C\cdot\Gamma \quad (\texttt{o@}l{:}\langle T_o\rangle\in\Gamma \vee \texttt{o@}l{:}\langle T_o,T_i\rangle\in\Gamma) \quad \vdash t{:}T_t \quad T_t \not\leq T_o \quad P \xrightarrow{\nu r\ \texttt{o@}l(\texttt{t})} P'}{B \triangleright_D P \xrightarrow{\texttt{error}} B \triangleright_D P}$$

We also know that $\mu = \texttt{error}$ and $S' = B \triangleright_D P$.

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{S}} B \triangleright_D P$ we know that it can only be typed using T-Service:

$$(\texttt{T-Service}) \quad \frac{D=\alpha_C\cdot\Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P} \tag{3.59}$$

From the grammar we know that P can have have three forms. From the semantics we know that P can not be $\mathbf{0}$. We therefore consider the two remaining possibilities:

Subcase $P = B_P \cdot t_P \cdot \tilde{m}_P$

Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{P}} P$ from 3.59 we know that it can only be typed using T-Process:

$$(\texttt{T-Process}) \quad \frac{\Gamma,\Gamma' \vdash_{\mathrm{B}} B_P \triangleright \Gamma'' \quad \Gamma,\Gamma' \vdash_{\mathrm{state}} t_P \quad \Gamma,\Gamma' \vdash_{\mathrm{queue}} \tilde{m}_P \quad \nexists \mathtt{o}.\ \mathtt{o@}l{:}\langle O\rangle \in \Gamma' \vee \mathtt{o}{:}\langle O\rangle \in \Gamma'}{\Gamma \vdash_{\mathrm{P}} B_P \cdot t_P \cdot \tilde{m}_P} \tag{3.60}$$

The only semantic rule which can be applied to premise $P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o@}l(\mathtt{t})} P'$ from E-Send-Lift is S-Send:

$$(\texttt{S-Send}) \quad \frac{B_P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o@}l(\mathtt{e})} B'_P}{B_P \cdot t_P \cdot \tilde{m}_P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o@}l(\mathtt{e}(t''))} B'_P \cdot t_P \cdot \tilde{m}_P}$$

From the form of S-Send we know that B_P takes a step of evaluation labeled $\boldsymbol{\nu} r\ \mathtt{o@}l(\mathtt{e})$. In order for B_P to take that step and to be well typed, $\Gamma, \Gamma' \vdash_{\mathrm{B}} B_P \triangleright \Gamma''$ must be typed with either T-SolResp-New, T-SolResp-Exists or T-Notification. We therefore consider all three cases:

Subcase T-SolResp-New

If $\Gamma, \Gamma' \vdash_{\mathrm{B}} B_P \triangleright \Gamma''$ is typed using T-SolResp-New then we see from the form of this rule that $T_e \leq T_o$ where T_e is the type of the message which is going to be send and T_o is the type of the message allowed to be send using the operation:

$$(\texttt{T-SolResp-New}) \quad \frac{\mathtt{o@}l{:}\langle T_o, T_i\rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathtt{o@}l(\mathtt{e})(\mathtt{x}) \triangleright upd(\Gamma, x, T_i)}$$

Since we have $T_e \leq T_o$ then we can not have $T_t \not\leq T_o$ where T_t is the type of the message which is going to be send. Therefore E-Send-Lift can not be applied to a well-typed Service.

Subcase T-SolResp-Exists

The proof is similar to the proof for subcase T-SolResp-New.

Subcase T-Notification

The proof is similar to the proof for subcase T-SolResp-New.

Subcase $P = P_1 \mid P_2$

Applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{P}} P$ from 3.59 we know that it can only be typed using T-Process-Par:

$$(\texttt{T-Process-Par}) \quad \frac{\Gamma \vdash_{\mathrm{P}} P_1 \quad \Gamma \vdash_{\mathrm{P}} P_2}{\Gamma \vdash_{\mathrm{P}} P_1 \mid P_2}$$

Applying the Inversion Lemma (Lemma 3.9) to the premises of this rule we know that there are three forms of respectively P_1 and P_2. For the form $P_3 \mid P_4$ this case loops. For the form $\mathbf{0}$, the other processes are investigated. At least one process in the service must

be of the form $B_P \cdot t_P \cdot \tilde{m}_P$ according to the semantics. Processes of this from are investigated in order to find the process which is sending the message. In that case subcase $P = B_P \cdot t_P \cdot \tilde{m}_P$ is followed.

Case E-Exec-Lift

The proof is similar to the proof for case E-Send-Lift.

For the rest of the base cases the proofs are straightforward. As an example we consider case S-Send-Lift:

$$(\texttt{S-Send-Lift}) \quad \frac{P \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)}} P'}{B \triangleright_D P \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)}} B \triangleright_D P'}$$

Since the label is $\boldsymbol{\nu} r\ \texttt{o@}l\texttt{(t)}$ it can not be any of the error labels.

There exists no inductive step cases at the service layer.

□

The property that a well-typed network can not take a transition labeled `error` is described in the following Lemma.

Lemma 3.23 (Lack of Errors at Network Layer)

Let $\Gamma \vdash_{\mathrm{N}} N$
then $N \xrightarrow{\mu} N'$
implies $\mu \neq \texttt{error}$

Proof.

Assume $\Gamma \vdash_{\mathrm{N}} N$ and $N \xrightarrow{\mu} N'$. The proof is done by induction on the derivation of $N \xrightarrow{\mu} N'$.

We now consider the base cases:

Case E-Comm

If the last rule in the derivation sequence is E-Comm, then from the form of this rule we know that $N = [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}$ for two services consisting of respectively behaviour B_1 and B_2, deployment part D_1 and D_2 and process P_1 and P_2. The services are running at respectively location l_1 and l_2:

$$\text{(E-Comm)}\quad \frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad (\texttt{o@}l_2 : \texttt{<}T_1\texttt{>} \in \Gamma_1 \vee \texttt{o@}l_2 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1) \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad (\texttt{o} : \texttt{<}T_2\texttt{>} \in \Gamma_2 \vee \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2) \quad T_1 \neq T_2 \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l_2(t)} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o}(t)} B_2 \triangleright_{D_2} P_2' \quad r \notin \texttt{cn}(S_1) \cup \texttt{cn}(S_2) \end{array}}{[B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2} \xrightarrow{\text{error}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}}$$

We also know that $\mu = \texttt{error}$ and $N' = \boldsymbol{\nu} r\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\text{N}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}$ we know that it can only be typed using T-Network:

$$\text{(T-Network)}\quad \frac{\begin{array}{c} \Gamma_1 \vdash_{\text{N}} [B_1 \triangleright_{D_1} P_1]_{l_1} \quad \Gamma_2 \vdash_{\text{N}} [B_2 \triangleright_{D_2} P_2]_{l_2} \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in \mathit{locs}([B_2 \triangleright_{D_2} P_2]_{l_2}).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin \mathit{locs}([B_1 \triangleright_{D_1} P_1]_{l_1}) \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in \mathit{locs}([B_1 \triangleright_{D_1} P_1]_{l_1}).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin \mathit{locs}([B_2 \triangleright_{D_2} P_2]_{l_2}) \\ \neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2) \end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{N}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}}$$

where $\Gamma = \Gamma_1 \cup \Gamma_2$. Since we from the form of this rule have $\forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1$ where $l \in \mathit{locs}([B_2 \triangleright_{D_2} P_2]_{l_2}).\, \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin \mathit{locs}([B_1 \triangleright_{D_1} P_1]_{l_1})$, then we can not have $(\texttt{o@}l_2 : \texttt{<}T_1\texttt{>} \in \Gamma_1 \vee \texttt{o@}l_2 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1)$ and therefore E-Comm can not be applied to a well-typed network.

Case E-Response

If the last rule in the derivation sequence is E-Response, then from the form of this rule we know that $N = \boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )$ for two services consisting of respectively behaviour B_1 and B_2, deployment part D_1 and D_2 and process P_1 and P_2. The services are restricted to channel name r and they runs at respectively location l_1 and l_2:

$$\text{(E-Response)}\quad \frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad \texttt{o@}l_1 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1 \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2 \quad T_2' \neq T_1' \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{(r, \texttt{o@}l_1)?t} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{(r, \texttt{o})!\,t} B_2 \triangleright_{D_2} P_2' \end{array}}{\boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ ) \xrightarrow{\text{error}} \boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )}$$

We also know that $\mu = \texttt{error}$ and $N' = \boldsymbol{\nu} r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )$. Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\text{N}} \boldsymbol{\nu} r (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )$ we know that it can only be typed using T-Restriction:

$$(\texttt{T-Restriction}) \quad \frac{\Gamma \vdash_{\mathrm{N}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}}{\Gamma \vdash_{\mathrm{N}} \nu r\ (\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )}$$

By applying the Inversion Lemma (Lemma 3.9) to premise $\Gamma \vdash_{\mathrm{N}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}$ from this rule we know that it can only be typed by rule T-Network. The rest of this proof is similar to the proof for case E-Comm.

For the rest of the base cases the proofs are straightforward.

For all the cases in the inductive step the proofs are similar. As an example we have shown case N-Par:

$$(\texttt{N-Par}) \quad \frac{N_1 \xrightarrow{\mu} N_1'}{N_1 \mid N_2 \xrightarrow{\mu} N_1' \mid N_2}$$

Applying the Inversion Lemma (Lemma 3.9) to $\Gamma \vdash_{\mathrm{N}} N_1 \mid N_2$ we know that it can only be typed using T-Network:

$$(\texttt{T-Network}) \quad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} N_1 \quad \Gamma_2 \vdash_{\mathrm{N}} N_2 \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\ \texttt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\texttt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \texttt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2}$$

where $\Gamma = \Gamma_1 \cup \Gamma_2$. The thesis follows from applying the induction hypothesis on premise $\Gamma_1 \vdash_{\mathrm{N}} N_1$ from T-Network applied on $\Gamma \vdash_{\mathrm{N}} N_1 \mid N_2$ and on premise $N_1 \xrightarrow{\mu} N_1'$ from N-Par, since the same label is used in premise and conclusion of N-Par. □

3.4.3 Type Safety

Type safety is that a well-typed statement is not able to take a step of evaluation labeled `error` at any point of its evaluation sequence. In section 3.4.1 we extended the semantics of Jolie with rules labeled `error` for services and networks. The new rules have in common that in order for a service or a network to take the transition, a message which type violates the type declarations for the operation it is communicated over must be send. In section 3.4.2 we formulated

that a well-typed service or network can not take a transition labeled `error`. The Lemmas presented in 3.4.2 can be seen as one step type safety. We will now extend these Lemmas to an evaluation sequence.

A well-typed network can never take a transition labeled `error`:

Theorem 3.24 (Type Safety)

$$\begin{array}{l} \textit{If } \Gamma \vdash N \\ \textit{and } N \rightarrow^* N' \\ \textit{then } N' \overset{error}{\nrightarrow} \end{array}$$

Proof.

By Lemma 3.23 we know that if a network is well typed, then it can not take a transition labeled `error`. By theorem 3.20 we know that if a network is well typed, then it is also well typed after taking a step of evaluation. Hence the thesis follows. □

We do not have to state the type safety theorem for services, because it is implicit in the type safety theorem for networks, since the combination of rule T-Network and T-Deployment requires that the services in a network are well typed in order for the network to be well typed.

$$\text{(T-Network)} \quad \frac{\begin{array}{c} \Gamma_1 \vdash_{\mathrm{N}} N_1 \quad \Gamma_2 \vdash_{\mathrm{N}} N_2 \\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\ \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\ \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\mathtt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \mathtt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2) \end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2}$$

$$\text{(T-Deployment)} \quad \frac{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P \quad l \notin locs(\Gamma)}{\{\mathtt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma\} \cup \{\mathtt{o@}l : \texttt{<}O\texttt{>} \mid \mathtt{o} : \texttt{<}O\texttt{>} \in \Gamma\} \vdash_{\mathrm{N}} [B \triangleright_D P]_l}$$

Chapter 4

Conclusion

type checker rejects networks of services in which a message is sent or received, where the message has a wrong type according to sender and receivers type specifications.

We have presented a type system for the core fragment of the Jolie language. The type system ensures that no message which has a wrong type according to the corresponding operation type specifications, is send nor received. To the best of the author's knowledge this is the first type system for the Jolie language with that purpose. Our type system guarantees the properties type preservation and type safety. The type preservation property ensures that if a network is well typed, then it is also well typed after taking a step of evaluation. The property type safety ensures that if a network is well typed then it is not able to take a step of evaluation labeled `error` at any point of its evaluation sequence.

4.1 Future Work

The future work proposals divides into three main areas: The area language extensions 4.1.1 considers extending the type system to handle language structures from other parts of the Jolie language than the core fragment. The area purpose

extension 4.1.2 considers extending the purpose of the type system. The area precision 4.1.3 considers reducing the approximations used in the type checker.

Extensions from the three areas can with benefit be combined.

4.1.1 Language Extentions

This thesis considers a fragment of Jolie which excludes recursive types, arrays, subtyping of basic types, faults and deployment instructions such as architectural composition. The following future work proposals are about extending the fragment of Jolie to the full Jolie language.

Recursive Types A future work is to extend this work with recursive types. This will primarily affect the fundament for the type system such as subtyping and the question of equality between types. It can be considered whether the recursive types shall be equi-recursive or iso-recursive, depending on which requirements for type equality are whished.

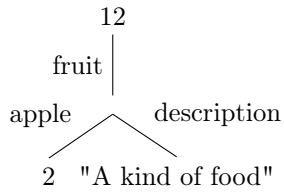
Subtyping of Basic Types The full Jolie language has subtyping of basic types. For instance is `int` a subtype of `long`. It can be added by writing the typing rules specific. The full Jolie language also contains the basic type `any`, which can be any of the basic types. It can be handled by adding a typing rule with the conclusion $bt \leq$ `any`.

In the work of adding subtyping of basic types the typing rules for assignment must then be updated to use the subtyping relation instead of the equality relation.

Fault handling An interesting extension of this work will be to consider fault handling, since faults alters the flow of a program.

Deployment Instructions An output port consists of a location, a protocol, operation type declarations and specifications for architectural compositions. This work can be extended to handle output ports and their corresponding settings. Of these settings only smart aggregation [PGG+12] is interesting, because it works as an proxy which is aware of types.

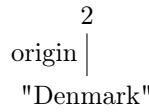
Arrays This work can be extended by adding arrays. In the full Jolie language each node of a data tree is considered an array. The root node is only allowed to be an array of exactly one element. Recall the example with the variable named amount from chapter 1:



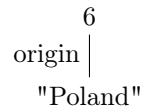
Each node is the first element of an array. It can also be accessed using index zero:

```
amount.fruit.apple[0].origin = "Denmark";
amount.fruit.apple[1] = 6
amount.fruit.apple[1].origin = "Poland"
```

The tree of amount.fruit.apple:



The tree of amount.fruit.apple[1]:



If an array is assigned more than one step from its current range, the intermediate array elements have no values.

Adding arrays doesn't change much in the definition of a type. Only the cardinality is affected. Where the cardinality in the core fragment of Jolie is an interval which at maximum can range to one, the cardinality of a fragment of Jolie including arrays allows for higher maxima:

$$C ::= \quad [\text{MIN, MAX}]$$

where MIN and MAX are integers and $0 \leq \text{MIN} \leq \text{MAX}$. The cardinality describes the maximum and minimum allowed number of elements of an array.

Elements of an array are allowed to have different types. How precise this is handled is a question about approximation. It can be handled by assigning each element the least supertype of all the types of the array elements. The type `any` from the full Jolie language is a supertype of the basic types. It must be introduced to the language of the type system in order to handle array elements with different basic types. The introduction of `any` is described in the Subtyping of Basic Types proposal.

As an array is extended, its cardinality must be updated too. Since the affected variable later may occur on either sides in the subtyping relation, it is better to be precise and update both boundaries in the cardinality, when an array is updated. Note that it should be considered whether the update of an array is an extension or an alternation of an already existing array element. This can be done looking at the cardinality. If it is an extension it shall be considered how far it is from the maximum array boundary in order to approximately determine the number of intermediate array elements.

4.1.2 Purpose Extensions

This thesis provides the theoretic foundation of type checking communication. Since it addresses the interaction between services it creates the foundation of type checking networks. The following future work proposals are about building on this foundation.

General Type Checking The aim of this work is to design a type system that type checks communication. In order achieve this goal the type system is also able to type check terms not involved in a communication. The type property proven is only regarding the goal. It can be reformulated to also include general type checking, e.g. that a condition must have type `bool`.

Combine with Type System from [MC11] Another interesting direction is to combine this work with the type system presented in [MC11], which focus on manipulating correlation sets.

Type Checking Data Flows The type system lays the foundation for type checking networks by checking that there is never send nor receive a message with the wrong type. By extending it to handle session types [HVK98], data flows in a network can be type checked.

4.1.3 Precision

The type system makes use of approximations which makes it reject safe networks. The approximations in the typing of statements which branches can be reduced by analyzing the expression and deduce the selected branch. It will not be duable in all situations, since part of the expression might be input from another service which output type declaration is not specific. Further precision in the typing of the parallel statement can be optained by analysing the parallel behaviours for dependencies.

Appendix A

Appendix

A.1 Semantics

A.1.1 Behavioural Layer

$$\text{(B-Choice)}\quad \frac{j\in J \quad \eta_j \xrightarrow{\mu} B'_j}{\sum_{i\in J}\,[\eta_i]\{B_i\} \xrightarrow{\mu} B'_j;B_j}$$

$$\text{(B-Struct)}\quad \frac{B_1\equiv B_2 \quad B_2 \xrightarrow{\mu} B'_2 \quad B'_1\equiv B'_2}{B_1 \xrightarrow{\mu} B'_1}$$

$$\text{(B-SolResp)}\quad \texttt{o@}l\texttt{(e)(x)} \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e)}} \mathsf{Wait}(r,\texttt{o@}l,\texttt{x}) \qquad \text{(B-Notification)}\quad \texttt{o@}l\texttt{(e)} \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e)}} \mathbf{0}$$

$$\text{(B-ReqResp)}\quad \texttt{o(x)(x')\{}B\texttt{\}} \xrightarrow{r:\texttt{o(x)}} \mathsf{Exec}(r,\texttt{o},\texttt{x'},B) \qquad \text{(B-OneWay)}\quad \texttt{o(x)} \xrightarrow{r:\texttt{o(x)}} \mathbf{0}$$

$$\text{(B-End-Exec)}\quad \mathsf{Exec}(r,\texttt{o},\texttt{x},\mathbf{0}) \xrightarrow{(r,\texttt{o})!\,x} \mathbf{0} \qquad \text{(B-Wait)}\quad \mathsf{Wait}(r,\texttt{o@}l,\texttt{x}) \xrightarrow{(r,\texttt{o@}l)?x} \mathbf{0}$$

$$\text{(B-Exec)}\quad \frac{B \xrightarrow{\mu} B'}{\mathsf{Exec}(r,\texttt{o},\texttt{x},B) \xrightarrow{\mu} \mathsf{Exec}(r,\texttt{o},\texttt{x},B')} \qquad \text{(B-Assign)}\quad \texttt{x = }e \xrightarrow{\texttt{x = }e} \mathbf{0}$$

$$\text{(B-Seq)}\quad \frac{B_1 \xrightarrow{\mu} B'_1}{B_1;B_2 \xrightarrow{\mu} B'_1;B_2} \qquad \text{(B-Par)}\quad \frac{B_1 \xrightarrow{\mu} B'_1}{B_1 \mid B_2 \xrightarrow{\mu} B'_1 \mid B_2}$$

$$\text{(B-If-Then)}\quad \frac{e(t)=true}{\texttt{if(}e\texttt{)}\ B_1\ \texttt{else}\ B_2 \xrightarrow{\textbf{read}\ t} B_1} \qquad \text{(B-If-Else)}\quad \frac{e(t)=false}{\texttt{if(}e\texttt{)}\ B_1\ \texttt{else}\ B_2 \xrightarrow{\textbf{read}\ t} B_2}$$

$$\text{(B-Iteration)}\quad \frac{e(t)=true}{\texttt{while(}e\texttt{)}\ \{B\} \xrightarrow{\textbf{read}\ t} B;\texttt{while(}e\texttt{)}\ \{B\}} \qquad \text{(B-No-Iteration)}\quad \frac{e(t)=false}{\texttt{while(}e\texttt{)}\ \{B\} \xrightarrow{\textbf{read}\ t} \mathbf{0}}$$

A.1.2 Service Layer

The structure of the whole service is omitted where it is irrelevant.

$$\text{(S-Get)}\quad \frac{B \xrightarrow{r:\texttt{o(x)}} B'}{B\cdot t\cdot(r,\texttt{o},t');\tilde{m} \xrightarrow{\tau} B'\cdot t\leftarrow_{\texttt{x}} t'\cdot\tilde{m}} \qquad \text{(S-Send)}\quad \frac{B \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e)}} B'}{B\cdot t\cdot\tilde{m} \xrightarrow{\nu r\ \texttt{o@}l\texttt{(e(}t\texttt{))}} B'\cdot t\cdot\tilde{m}}$$

$$\text{(S-Exec)}\quad \frac{B \xrightarrow{(r,\texttt{o})!\,x} B'}{B\cdot t\cdot\tilde{m} \xrightarrow{(r,\texttt{o})!\,x(t)} B'\cdot t\cdot\tilde{m}} \qquad \text{(S-Wait)}\quad \frac{B \xrightarrow{(r,\texttt{o@}l)?x} B'}{B\cdot t\cdot\tilde{m} \xrightarrow{(r,\texttt{o@}l)?t'} B'\cdot t\leftarrow_{\texttt{x}} t'\cdot\tilde{m}}$$

$$\text{(S-Assign)}\quad \frac{B \xrightarrow{\texttt{x = }e} B'}{B\cdot t\cdot\tilde{m} \xrightarrow{\tau} B'\cdot t\leftarrow_{\texttt{x}} e(t)\cdot\tilde{m}} \qquad \text{(S-Read)}\quad \frac{B \xrightarrow{\textbf{read}\ t} B'}{B\cdot t\cdot\tilde{m} \xrightarrow{\tau} B'\cdot t\cdot\tilde{m}}$$

(S-Corr) $$\frac{D{=}\alpha_C{\cdot}\Gamma \quad t',\mathtt{o} \vdash_{\alpha_C} t \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'} \leq T_i}{B \triangleright_D P \mid B'{\cdot}t{\cdot}\tilde{m} \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}(t')} B \triangleright_D P \mid B'{\cdot}t{\cdot}\tilde{m}\{r,\mathtt{o},t'\}}$$

(S-Start) $$\frac{D{=}\alpha_C{\cdot}\Gamma \quad t,\mathtt{o} \not\vdash_{\alpha_C} P \qquad B \xrightarrow{r:\mathtt{o}(\mathtt{x})} B' \qquad t'{=}\mathsf{init}(t,\mathtt{o},\alpha_C) \quad (\mathtt{o}{:}\langle T_i\rangle{\in}\Gamma \vee \mathtt{o}{:}\langle T_i,T_o\rangle{\in}\Gamma) \quad \vdash t'{:}T_{t'} \quad T_{t'} \leq T_i}{B \triangleright_D P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}(t)} B \triangleright_D P \mid B'{\cdot}t_\perp \leftarrow_{\mathtt{x}} t \leftarrow_{\mathtt{csets}} t'{\cdot}\epsilon}$$

(S-Tau) $$\frac{P \xrightarrow{\tau} P'}{B \triangleright_D P \xrightarrow{\tau} B \triangleright_D P'}$$

(S-Send-Lift) $$\frac{P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}@l(\mathtt{t})} P'}{B \triangleright_D P \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}@l(\mathtt{t})} B \triangleright_D P'}$$

(S-Exec-Lift) $$\frac{P \xrightarrow{(r,\mathtt{o})!\ t} P'}{B \triangleright_D P \xrightarrow{(r,\mathtt{o})!\ t} B \triangleright_D P'}$$

(S-Wait-Lift) $$\frac{P \xrightarrow{(r,\mathtt{o}@l)?t} P'}{B \triangleright_D P \xrightarrow{(r,\mathtt{o}@l)?t} B \triangleright_D P'}$$

(S-Par) $$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$\mathsf{init}(t,\mathtt{o},\alpha_C) = \begin{cases} t_\perp \twoheadleftarrow_{\mathtt{p}_1} f(\mathtt{p}_1)(t) \ldots \twoheadleftarrow_{\mathtt{p}_n} f(\mathtt{p}_n)(t) & \text{if } \alpha_C(\mathtt{o}) = (\{\mathtt{p}_1,\ldots,\mathtt{p}_n\},f) \\ t_\perp & \text{if } \mathtt{o} \notin \mathit{Dom}(\alpha_C) \\ \text{undefined} & \text{otherwise} \end{cases}$$

A.1.3 Network Layer

(N-Comm) $$\frac{S_1 \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}@l_2(t)} S_1' \quad S_2 \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}(t)} S_2' \quad r \notin \mathtt{cn}(S_1) \cup \mathtt{cn}(S_2)}{[S_1]_{l_1} \mid [S_2]_{l_2} \xrightarrow{\tau} \boldsymbol{\nu} r\ (\ [S_1']_{l_1} \mid [S_2']_{l_2}\ )}$$

(N-Response) $$\frac{S_1 \xrightarrow{(r,\mathtt{o}@l_1)?t} S_1' \quad S_2 \xrightarrow{(r,\mathtt{o})!\ t} S_2'}{\boldsymbol{\nu} r\ (\ [S_1]_{l_1} \mid [S_2]_{l_2}\ ) \xrightarrow{\tau} [S_1']_{l_1} \mid [S_2']_{l_2}}$$

(N-Par) $$\frac{N_1 \xrightarrow{\mu} N_1'}{N_1 \mid N_2 \xrightarrow{\mu} N_1' \mid N_2}$$

(N-Tau) $$\frac{S \xrightarrow{\tau} S'}{[S]_l \mid N \xrightarrow{\tau} [S']_l \mid N}$$

(N-Struct) $$\frac{N_1 \equiv N_2 \quad N_2 \xrightarrow{\mu} N_2' \quad N_1' \equiv N_2'}{N_1 \xrightarrow{\mu} N_1'}$$

(N-Restriction) $$\frac{N \xrightarrow{\tau} N'}{\boldsymbol{\nu} r\ (\ N\ ) \xrightarrow{\tau} \boldsymbol{\nu} r\ (\ N'\ )}$$

A.1.4 Error Rules

$$\frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad (\texttt{o@}l_2 : \texttt{<}T_1\texttt{>} \in \Gamma_1 \vee \texttt{o@}l_2 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1) \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad (\texttt{o} : \texttt{<}T_2\texttt{>} \in \Gamma_2 \vee \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2) \quad T_1 \neq T_2 \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l_2(t)} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{\boldsymbol{\nu} r\ \texttt{o}(t)} B_2 \triangleright_{D_2} P_2' \quad r \notin \texttt{cn}(S_1) \cup \texttt{cn}(S_2) \end{array}}{[B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2} \xrightarrow{\texttt{error}} [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}} \quad \texttt{(E-Comm)}$$

$$\frac{D = \alpha_C \cdot \Gamma \quad (\texttt{o@}l : \texttt{<}T_o\texttt{>} \in \Gamma \vee \texttt{o@}l : \texttt{<}T_o, T_i\texttt{>} \in \Gamma) \quad \vdash t : T_t \quad T_t \not\leq T_o \quad P \xrightarrow{\boldsymbol{\nu} r\ \texttt{o@}l\,\texttt{(t)}} P'}{B \triangleright_D P \xrightarrow{\texttt{error}} B \triangleright_D P} \quad \texttt{(E-Send-Lift)}$$

$$\frac{D = \alpha_C \cdot \Gamma \quad \texttt{o@}l : \texttt{<}T_i, T_o\texttt{>} \in \Gamma \quad \vdash t : T_t \quad T_t \not\leq T_o \quad P \xrightarrow{(r,\texttt{o})!\,t} P'}{B \triangleright_D P \xrightarrow{\texttt{error}} B \triangleright_D P} \quad \texttt{(E-Exec-Lift)}$$

$$\frac{\begin{array}{c} D_1 = \alpha_C \cdot \Gamma_1 \quad \texttt{o@}l_1 : \texttt{<}T_1, T_1'\texttt{>} \in \Gamma_1 \\ D_2 = \alpha_C' \cdot \Gamma_2 \quad \texttt{o} : \texttt{<}T_2, T_2'\texttt{>} \in \Gamma_2 \quad T_2' \neq T_1' \\ B_1 \triangleright_{D_1} P_1 \xrightarrow{(r,\texttt{o@}l_1)?t} B_1 \triangleright_{D_1} P_1' \quad B_2 \triangleright_{D_2} P_2 \xrightarrow{(r,\texttt{o})!\,t} B_2 \triangleright_{D_2} P_2' \end{array}}{\boldsymbol{\nu} r\,(\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ ) \xrightarrow{\texttt{error}} \boldsymbol{\nu} r\,(\ [B_1 \triangleright_{D_1} P_1]_{l_1} \mid [B_2 \triangleright_{D_2} P_2]_{l_2}\ )} \quad \texttt{(E-Response)}$$

A.2 Type System

A.2.1 Subtyping

$$\frac{BT_1 \leq BT_2 \quad CTL_1 \leq CTL_2}{BT_1\{CTL_1\} \leq BT_2\{CTL_2\}} \quad \texttt{(ST-T)}$$

$$\frac{BT_1 = BT_2}{BT_1 \leq BT_2} \quad \texttt{(ST-BT)}$$

$$\frac{\begin{array}{c} dom(CTL_1) \subseteq dom(CTL_2) \\ \forall x \in dom(CTL_2)\,.\,CTL_2(x) =< C_2, T_2 > \wedge \\ CTL_1(x, T_2) =< C_1, T_1 > \wedge C_1 \leq C_2 \wedge T_1 \leq T_2 \end{array}}{CTL_1 \leq CTL_2} \quad \texttt{(ST-CTL)}$$

$$\frac{MIN_2 \leq MIN_1 \quad MAX_1 \leq MAX_2}{[MIN_1, MAX_1] \leq [MIN_2, MAX_2]} \quad \texttt{(ST-C)}$$

$$\frac{BT_1 \leq BT_2 \quad \forall x \in dom(CTL)\,.\,CTL(x) =< C, T > \wedge [0,0] \leq C}{BT_1 \leq BT_2\{CTL\}} \quad \texttt{(ST-BT-T)}$$

A.2.2 Typing Rules at Behavioural Layer

(T-Nil) $\dfrac{}{\Gamma \vdash_{\mathrm{B}} \mathbf{0} \triangleright \Gamma}$

(T-Assign-New) $\dfrac{\Gamma \vdash e{:}T_e \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathtt{x = } e \triangleright upd(\Gamma, x, bt(T_e))}$

(T-Assign-Exists) $\dfrac{\Gamma \vdash e{:}T_e \quad x{:}T_x \in \Gamma \quad bt(T_e){=}bt(T_x)}{\Gamma \vdash_{\mathrm{B}} \mathtt{x = } e \triangleright \Gamma}$

(T-If-Then-Else) $\dfrac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{B}} B_2 \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \mathtt{if}(e)\ B_1\ \mathtt{else}\ \ B_2 \triangleright \Gamma'}$

(T-Choice) $\dfrac{\forall j \in J\ .\ \Gamma \vdash_{\mathrm{B}} \eta_j ; B_j \triangleright \Gamma'}{\Gamma \vdash_{\mathrm{B}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \Gamma'}$

(T-Par) $\dfrac{\Gamma_1 \vdash_{\mathrm{B}} B_1 \triangleright \Gamma_1' \quad \Gamma_2 \vdash_{\mathrm{B}} B_2 \triangleright \Gamma_2' \quad Roots(\Gamma_1') \cap Roots(\Gamma_2') = \emptyset}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{B}} B_1 \mid B_2 \triangleright \Gamma_1' \uplus \Gamma_2'}$

(T-Seq) $\dfrac{\Gamma \vdash_{\mathrm{B}} B_1 \triangleright \Gamma' \quad \Gamma' \vdash_{\mathrm{B}} B_2 \triangleright \Gamma''}{\Gamma \vdash_{\mathrm{B}} B_1 ; B_2 \triangleright \Gamma''}$

(T-While) $\dfrac{\Gamma \vdash e{:}bool \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathtt{while}(e)\ \{B\} \triangleright \Gamma}$

(T-Notification) $\dfrac{\mathtt{o@}l{:}\langle T_o \rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathtt{o@}l\mathtt{(e)} \triangleright \Gamma}$

(T-SolResp-New) $\dfrac{\mathtt{o@}l{:}\langle T_o, T_i \rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathtt{o@}l\mathtt{(e)(x)} \triangleright upd(\Gamma, x, T_i)}$

(T-SolResp-Exists) $\dfrac{\mathtt{o@}l{:}\langle T_o, T_i \rangle \in \Gamma \quad \Gamma \vdash e{:}T_e \quad T_e \leq T_o \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathtt{o@}l\mathtt{(e)(x)} \triangleright \Gamma}$

(T-OneWay-New) $\dfrac{\mathtt{o}{:}\langle T_i \rangle \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathtt{o(x)} \triangleright upd(\Gamma, x, T_i)}$

(T-OneWay-Exists) $\dfrac{\mathtt{o}{:}\langle T_i \rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathtt{o(x)} \triangleright \Gamma}$

(T-ReqResp-New) $\dfrac{\mathtt{o}{:}\langle T_i, T_o \rangle \in \Gamma \quad x \notin \Gamma \quad upd(\Gamma, x, T_i) \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x'{:}T_{x'} \in \Gamma' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathtt{o(x)(x')}\ \{B\} \triangleright \Gamma'}$

(T-ReqResp-Exists) $\dfrac{\mathtt{o}{:}\langle T_i, T_o \rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x'{:}T_{x'} \in \Gamma' \quad T_{x'} \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathtt{o(x)(x')}\ \{B\} \triangleright \Gamma'}$

A.2.2.1 Run-Time Types at Behavioural Layer

(T-Wait-New) $$\frac{\mathtt{o@}l{:}\langle T_o,T_i\rangle \in \Gamma \quad x \notin \Gamma}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r,\mathtt{o@}l,\mathtt{x}) \triangleright upd(\Gamma,x,T_i)}$$

(T-Wait-Exists) $$\frac{\mathtt{o@}l{:}\langle T_o,T_i\rangle \in \Gamma \quad x{:}T_x \in \Gamma \quad T_i \leq T_x}{\Gamma \vdash_{\mathrm{B}} \mathsf{Wait}(r,\mathtt{o@}l,\mathtt{x}) \triangleright \Gamma}$$

(T-Exec) $$\frac{\mathtt{o}{:}\langle T_i,T_o\rangle \in \Gamma \quad \Gamma \vdash_{\mathrm{B}} B \triangleright \Gamma' \quad x{:}T_x \in \Gamma' \quad T_x \leq T_o}{\Gamma \vdash_{\mathrm{B}} \mathsf{Exec}(r,\mathtt{o},\mathtt{x},B) \triangleright \Gamma'}$$

A.2.3 Typing Rules at Service Layer

(T-BSL-Nil) $$\frac{}{\Gamma \vdash_{\mathrm{BSL}} \mathbf{0} \triangleright \Gamma}$$

(T-BSL-Choice) $$\frac{\forall j \in J \;.\; \Gamma \vdash_{\mathrm{B}} \eta_j;B_j \triangleright \Gamma_j \quad \nexists \mathrm{T}_k,\mathrm{T}_l \,.\, x{:}\mathrm{T}_k \in \bigcup \Gamma_j \wedge x{:}\mathrm{T}_l \in \bigcup \Gamma_j \wedge \mathrm{T}_k \neq \mathrm{T}_l}{\Gamma \vdash_{\mathrm{BSL}} \sum_{i \in J} [\eta_i]\{B_i\} \triangleright \bigcup \Gamma_{j \in J}}$$

(T-Process-Nil) $$\frac{}{\Gamma \vdash_{\mathrm{P}} \mathbf{0}}$$

(T-Process) $$\frac{\Gamma,\Gamma' \vdash_{\mathrm{B}} B \triangleright \Gamma'' \quad \Gamma,\Gamma' \vdash_{\mathrm{state}} t \quad \Gamma,\Gamma' \vdash_{\mathrm{queue}} \tilde{m} \quad \nexists \mathtt{o}.\, \mathtt{o@}l{:}\langle O\rangle \in \Gamma' \vee \mathtt{o}{:}\langle O\rangle \in \Gamma'}{\Gamma \vdash_{\mathrm{P}} B{\cdot}t{\cdot}\tilde{m}}$$

(T-Process-Par) $$\frac{\Gamma \vdash_{\mathrm{P}} P_1 \quad \Gamma \vdash_{\mathrm{P}} P_2}{\Gamma \vdash_{\mathrm{P}} P_1 \mid P_2}$$

(T-Service) $$\frac{D{=}\alpha_C{\cdot}\Gamma \quad \Gamma \vdash_{\mathrm{BSL}} B \triangleright \Gamma' \quad \Gamma \vdash_{\mathrm{P}} P}{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P}$$

A.2.4 Typing Rules at Network Layer

(T-Network-Nil) $\dfrac{}{\Gamma \vdash_{\mathrm{N}} \mathbf{0}}$

(T-Deployment) $\dfrac{\Gamma \vdash_{\mathrm{S}} B \triangleright_D P \quad l \notin locs(\Gamma)}{\{\mathtt{o@}l' : \texttt{<}O\texttt{>} \in \Gamma\} \cup \{\mathtt{o@}l : \texttt{<}O\texttt{>} \mid \mathtt{o} : \texttt{<}O\texttt{>} \in \Gamma\} \vdash_{\mathrm{N}} [B \triangleright_D P]_l}$

(T-Network) $\dfrac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{N}} N_1 \quad \Gamma_2 \vdash_{\mathrm{N}} N_2 \\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \text{ where } l \in locs(N_2).\ \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \wedge l \notin locs(N_1) \\ \forall \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_2 \text{ where } l \in locs(N_1).\ \mathtt{o@}l : \texttt{<}O\texttt{>} \in \Gamma_1 \wedge l \notin locs(N_2) \\ \neg(\mathtt{o@}l : \texttt{<}O_1\texttt{>} \in \Gamma_1 \wedge \mathtt{o@}l : \texttt{<}O_2\texttt{>} \in \Gamma_2 \wedge O_1 \neq O_2)\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{N}} N_1 \mid N_2}$

(T-Restriction) $\dfrac{\Gamma \vdash_{\mathrm{N}} N}{\Gamma \vdash_{\mathrm{N}} \boldsymbol{\nu} r\ N}$

Bibliography

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: a calculus for service oriented computing. In *Service-Oriented Computing–ICSOC 2006*, pages 327–338. Springer, 2006.

[HVK98] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer, 1998.

[Jol] Jolie. Jolie programming language - official website. http://www.jolie-lang.org/.

[MC11] Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In *Service-Oriented Computing*, pages 125–141. Springer, 2011.

[MGZ] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie.

[Mon10] Fabrizio Montesi. Jolie: a service-oriented programming language. *Master's thesis, University of Bologna, Department of Computer Science*, 2010.

[PGG+12] Mila Dalla Preda, Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Interface-based service composition with aggregation. In *ESOCC*, pages 48–63, 2012.

[TCBM06] WT Tsai, Yinong Chen, Gary Bitter, and Dorina Miron. Introduction to service-oriented computing. *Arizona State University*, 2006.

[W3C] W3C. Web services architecture. http://www.w3.org/tr/ws-arch/.