Imperial College London

Department of Computing

A Session Type Discipline for Event Driven Programming Models

by

Dimitrios Kouzapas

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2009

Acknowledgments

I would like to acknowledge the contribution of my supervisor Dr Nobuko Yoshida for her guidance and the time she devoted for this work. I would also like to acknowledge Raymont Hu for our cooperation in the development of the algebra. Last Dimitris Mostrous helped me to better understand background notions around the subject.

Abstract

This work develops a session type discipline for event driven programming models. Event models are used to describe asynchrony in concurrent systems. Session types is a typing system for π calculus developed for sound and safe process interaction. The studying of events through a session type definition presents great interest for concurrent systems. For the purpose of this thesis event driven models are studied in order to find their key properties that can be used for a theoretical definition of an event driven model. A process algebra based on π calculus is then developed to describe event driven programming. This algebra is based on asynchronous communication and it is extended with constructs that handle events. Some novelties are also present such as the asynchronous session creation. A proper session typing system, able to describe the nature of events, is defined along this process algebra. Through examples the algebra and the typing system are practically evaluated. Finally Subject Reduction and Type Safety are formed and proved to provide soundness to the system. The discussion ends by studying deadlock processes and suggests ways for avoiding these situations.

Contents

Chapter 1

Introduction

As computer scientists we seek ways to further improve the systems exist today. Concurrency seems a promising way for increasing the efficiency off systems and many resources of research are spent towards this way. Researchers are trying to make their best out of parallel systems in almost all computation levels of abstraction.

Computer architecture scientists seek to design and build parallel platforms and expose the Instruction Set Architecture parallelism and data dependency parallelism to the machine. Graphic processors and other massively parallel processors are studied to be applied for more general applications. Compilers use techniques that reveal parallelism of a program and translate to parallel low level instructions. Operating systems consider more and more concurrency and multiple cores, processors and generally distributed resources in their kernel and platform design. Application engineering follows the object oriented approach which can be thought as a distributed system due to the extension of the object oriented framework to support concurrency. Finally network applications are heavily relied on concurrency and parallel principles. Whether we have a server - client model or a peer to peer model communication and concurrency is obvious.

In all computation layers we have a turn towards concurrency. Scientists are interested in exploiting concurrency and are interested in revealing parallelism by studying the concurrent properties of each layer. This observation gives rise to the need to formally describe concurrent systems in a strict mathematical framework.

A theory around concurrency and the study of that theory will reveal the potential and the limitation of distributed concurrent systems. This would be a very important tool in the hands of scientists because they can better understand concurrency and have a more concrete intuition about their research targets. The current trend towards parallelism gives a strong motive to develop a theory that defines concurrency.

Theoreticians tried over the years to formally describe the essence of concurrency. Formalising concurrency helps to study concurrent properties and capabilities and build complete systems. By complete systems we mean that we want correct, safe, efficient descriptions of systems. Systems where developed over the years and the ongoing research is based on that ideas. CCS [18], π calculus [19] and other process algebras were developed and used for theoretical research around the subject.

This project will address to the subject of concurrency taking a strict theoretical approach. We are particularly interested in developing an Event Driven Model using the Session Types Discipline [15], [25].

The event driven model is a model used to describe concurrency. Asynchronous communication is a property of message passing concurrent systems. The event driven model was developed so we can program under asynchronous parallel conditions. It is very interesting to define such a model in theoretic terms. We can say that the event driven model uses concurrency to cope with concurrency. This observation makes gives another motive for this work.

Session types where developed as a typing system for π calculus. π calculus is a computational model that uses notions around parallelism for its definition. It is used to specify message passing communication and the interaction between distributed processes. It is widely used to study

concurrency in a theoretical level.

Session types is a promising tool that tries to type communication and provides safety and soundness in the transitions of π calculus. Interaction between parties is done through channels called sessions. The communication sequence is followed by the typing system and a session type is constructed. By defining relations between session types we can provide safety to a system.

Communication safety is very important. The first reason is that programmers are subject to errors. A typing system should reveal the programming inconsistency and inform the programmer. This way consistent programs are built. Following the discussion we can see that data exchange is no more considered as streaming of typeless bytes, as other message passing interfaces consider, e.g the Message Passing Interface [21]. Instead each send/receive value has a type and it is considered at compile time. This property enhances security and strengthens the defence to attacks such as buffer overflow attack.

In the whole project we follow the standard approach for defining and studying the properties of a computational system that is checked through a typing system. A π calculus algebra is defined, extended with constructs that allow us to model event driven programming. The reduction semantics are given next. This is enough for the specification of the computational model. Session types need to develop a type discipline that is composed by the definitions around types and a typing system that runs along the tree of a π calculus event program and analyse its definition.

After the session type definition is given we define notions and form some basic theorems that satisfy the properties of soundness for a theoretical system. The proves of these theorems are given in a high level of detail.

Before we proceed with the definition of the session computational model a discussion about the nature of the event driven programming model is done. We recognize the key aspects of event driven models and we make our design choices. The reason for this choices and their π calculus definition equivalence are explained through the work.

All the above are done with the basic guideline that we want to keep the definition simple and yet powerful enough to express every aspect of an event driven model. We do not claim that we have model everything around the event driven model but this should be a complete study around the notions we try to define.

Chapter 2 gives background information about session types. In chapter 3 we have a discussion about events and there is a try to recognize the key properties and categories of event driven programming. A presentation of an event driven architecture is also given. Chapter 4 gives the notions that we are trying to define, a complete definition of the computational model, complete definitions around session types, a static and a runtime typing system. Chapter 5 gives program examples that are specified and typed using the definition of chapter 4. Useful conclusions can be extracted from this chapter. Chapter 6 does a detailed study of the properties of the typing system. It forms and proves a Subject Reduction and a Type Safety theorem along with all the supporting definitions and lemmas used for the proof. The discussion in this chapter finishes by a deadlock scenarion and a proposal for ways to avoid these situations. Chapter 7 gives the conclusions from the whole project and finally it gives guidelines for potential future work.

Chapter 2

Session Types

Every computational model is described by a BNF syntax. This syntax can be abstracted as a tree; with each node representing a computational instance along is sub-tree. A broad spectrum of constructs have been defined through programming language syntax that become more and more complicated as we proceed higher in the levels of abstraction. Higher level languages also use more complex primitive structures for their functionality. Core languages such as the core λ calculus [4] are used for description of these constructs in the lowest possible level. But syntax alone is not enough to completely describe a computational model or a programming language.

For example consider the evaluation of the if - else structure. If - else structure is a basic abstraction used for controlling a programs derivation. If the control expression of this structure is evaluated to something other than true or false then the derivation can stuck as more syntax trees do not specify this situation. To handle these kinds of problems another tree should be defined on the syntax tree called a typing tree or a type system. Type systems are used to statically detect errors that are not detected by the syntax of the model but are detected in the interaction between primitive constructs or constructs and primitive or more complex data structures.

A correct type system does not contain a computational specification that stucks at any given point of the computation. This is always done by reference to the types and the constructs of the syntax of the model. This property is very important when designing type systems and should be considered for correct type systems.

The π calculus formalism was developed as a core language that tries to describe primitives such as communication and concurrency. Concurrent programming is becoming more and more important as systems evolve.

As with all computational models, typing systems were developed for π calculus. *Session types* was initially introduced around a decade ago in [15]. In this work a typing system for π calculus [19] is presented that is based on entities called types. A session is a channel between two process entities. The interaction between the channels through a session is grasped by session types, from the time the session is created until the communication pattern ends. Essentially session types handle communication as a construct and along its structure it builds types. This communication is restricted on session channels that are created exclusively by a program. The basic idea is to be able to combine processes that have consistent session communication pattern. The execution should not get stuck in any part of the derivation and should result in a terminal state.

Next we become more specific about π calculus and Session Types.

2.1 π - Calculus

A description of the original π calculus formalism can be found in [19]. This model tries to model the behaviour of distributed systems. One of the first models that tried to specify model systems was the CCS process algebra [18]. π calculus is a process algebra based on the CCS process algebra.

Intuitively the basic construct of π calculus is a name. Processes are constructed over names and computational transitions consider the exchange of values and names through names. We construct processes by action continuation and by parallel composition. Actions take place on

names and parallel composition considers continuation of names in parallel. A transition action happens between parallel processes. This kind of actions can be thought as the communication or message passing between processes. We will become more formal and thus more specific below.

After the development of the original π calculus theory many algebras that have the same basis were developed. Session types also developed a π calculus variation that has as a central entity the session. A session is a name that is created through shared name interaction and has the property that is private for two parallel composed processes. Through this session message passing communication takes place. The two parallel processes implement a communication message passing protocol through their private session.

Session types originally described in [15] where the π calculus algebra is defined. Also the session types discipline is defined and presented through the paper. A revised version of session types can be found in [25]. This paper gives a more complete theory about session types. Specifically it defines the process algebra and gives a typing system and proves theorems such as Subject Reduction and Type Safety that complete the soundness of session types. Next it gives a variation of the session types systems and again it proves soundness for that system.

Variation on session types system shows that session types is a robust system that can be used to specify a wide variety of communication scenarios, application and generally communication systems. Currently there is a lot of research to describe different aspects of concurrent programming under the session type discipline.

Following we will give the syntax and the reduction rules of the revised session types [25]. In the next session we will give the type syntax and the type system of session types. The rules here will help the reader to understand clearly the π calculus formalism.

For the algebra that follows consider that names range over $a, b, x, y, z, \ldots$. Channels over k, k', constants range over $c, c', \ldots$, labels range over $l, l', \ldots$, process variables over $\mathbf{X}, \mathbf{Y}, \ldots$. $u, u', \ldots$ denote names and channels. Processes range over $P, Q, R, \ldots$ and expressions over e, e'. In figure 2.1 we can see the construction of processes.

Briefly we have in order of appearance the two constructs over names that create a new session. Output and input session actions follow. After that the selection and branching actions use labelling to control derivation. Throwing and catching models channel delegation with the former sending and the latter receiving. Following we have the standard control construct, parallel, inaction process and restriction follow the standard notions in π calculus. Recursion is given by the next rule where series of process variables are defined as processes. Finally there is the process variable construct. Expression can be constants or operators on expressions.

In figure 2.2 the reduction rules are given. In order of appearance the first rule considers session creation that is done on names. Communication on a session is done by a correspondence of sending receiving actions. The same case applies for labelling. One process sends a label and the other proceeds as defined. The communication case continues when delegating sessions in the next rule. Throwing models sending, catching models receiving. Next two rules are standard to describe selection through expression reduction. Next rule gives process variable instantiation. A process variable is instantiated by its body and a variable substitution is done. Next four rules are standard for π calculus and complete the reduction system.

A version of π calculus called asynchronous π calculus is described in [6]. This algebra is similar to the original π calculus description with the difference in the output prefix. Output prefix is not followed by a process continuation. This way we can implement asynchrony. A process that wants to send a message simply leaves a parallel composition of a sending prefix and the process continuation. Communication can happen any time with the receiving process interacting with the parallel output prefix.

In this work we consider asynchrony in a different way that it is described in asynchronous π calculus. Asynchrony is handled using First In First Out queue configurations as a structure defined in the algebra. Sending and receiving processes interact with this structure to send and receive messages to/from it.

We can see a relevant discussion about this structures in [5]. This work develops a π calculus algebra that tries to model asynchronous π calculus. In this algebra the authors define data

P, Q	$::=$	request $a(k)$ in P	(session request)
	$\mid$	accept $a(k)$ in P	(session acceptance)
	$\mid$	$k![\tilde{e}]; P$	(data sending)
	$\mid$	$k?(x)$ in P	(data reception)
	$\mid$	$k \triangleleft l; P$	(label selection)
	$\mid$	$k \triangleright \{l_i : P_i\}_{i \in I}$	(label branching)
	$\mid$	throw $k[k']; P$	(channel sending)
	$\mid$	catch $k(k')$ in P	(channel reception)
	$\mid$	`if` e `then` P `else` Q	(conditional branch)
	$\mid$	$P \mid Q$	(parallel composition)
	$\mid$	$(\boldsymbol{\nu}\, u)\ P$	(name/channel hiding)
	$\mid$	`def` D `in` P	(recursion)
	$\mid$	$\mathbf{X}[\tilde{e}\tilde{k}]$	(process variables)
	$\mid$	inact	(inaction)
e	$::=$	c	(constant)
	$\mid$	$e + e' \mid e - e' \mid not(e) \mid \ldots$	(operators)
D	$::=$	$\mathbf{X}_1(\tilde{x}_1\tilde{k}_1) = P_1 and \ldots and \mathbf{X}_n(\tilde{x}_n\tilde{k}_1) = P_n$	(recursion declaration)

Figure 2.1: Syntax

accept $a(k)$ in $P_1 \mid$ request $a(k)$ in $P_2 \rightarrow (\boldsymbol{\nu}\, k)\ (P_1 \mid P_2)$	(LINK)
$k![\tilde{e}]; P_1 \mid k(\tilde{x})$ in $P_2 \rightarrow P_1 \mid P_2\{\tilde{c}/\tilde{x}\} \quad (\tilde{e} \downarrow \tilde{c})$	(COM)
$k \triangleleft l_k; ; P \mid k \triangleright \{l_i : P_i\}_{i \in I} \rightarrow P \mid P_k \quad (k \in I)$	(LABEL)
throw $k[k']; P_1 \mid$ catch $k(k')$ in $P_2 \rightarrow P_1 \mid P_2$	(PASS)
`if` e `then` P `else` $Q \rightarrow P \quad (e \downarrow \text{true})$	(IF1)
`if` e `then` P `else` $Q \rightarrow Q \quad (e \downarrow \text{false})$	(IF2)
`def` D `in` $(\mathbf{x}[\tilde{e}\tilde{x}] \mid Q) \rightarrow$ `def` D `in` $(P[\tilde{c}\tilde{x}] \mid Q) \quad (e \downarrow c, \mathbf{X}[\tilde{x}\tilde{k} = P \in D]$	(DEF)
$P \rightarrow P' \Rightarrow (\boldsymbol{\nu}\, u)\ P \rightarrow (\boldsymbol{\nu}\, u)\ P'$	(SCOP)
$P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q$	(PAR)
$P \rightarrow P' \Rightarrow$ `def` D `in` $P \rightarrow$ `def` D `in` P'	(DEFIN)
$P \equiv P'$ and $P \rightarrow Q'$ and $Q' \equiv Q \Rightarrow P \rightarrow Q$	(STR)

Figure 2.2: Reduction

$$
\begin{array}{rcl}
\text{Sort } S & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \langle \alpha, \overline{\alpha} \rangle \\
\text{Type } \alpha & ::= & ?[\tilde{S}]; \alpha \mid ?[\alpha]; \beta \mid \&\{l_1 : \alpha_1 \ldots l_n : \alpha_n\} \mid \mathtt{end} \mid \bot \mid \\
 & & ![\tilde{S}]; \alpha \mid ![\alpha]; \beta \mid \oplus\{l_1 : \alpha_1 \ldots l_n : \alpha_n\} \mid \mathbf{t} \mid \mu \mathbf{t}.\alpha
\end{array}
$$

$$
\begin{array}{ccc}
\overline{![\tilde{S}]; \alpha} = ?[\tilde{S}]; \overline{\alpha} & \overline{\oplus\{l_i : \alpha_i\}_{i \in I}} = \&\{l_i : \overline{\alpha_i}\}_{i \in I} & \overline{![\alpha]; \beta} = ?[\alpha]; \overline{\beta} \\
\overline{?[\tilde{S}]; \alpha} = ![\tilde{S}]; \overline{\alpha} & \overline{\&\{l_i : \alpha_i\}_{i \in I}} = \oplus\{l_i : \overline{\alpha_i}\}_{i \in I} & \overline{?[\alpha]; \beta} = ![\alpha]; \overline{\beta} \\
\overline{\mathtt{end}} = \mathtt{end} & \overline{\mu \mathbf{t}.\alpha} = \mu \mathbf{t}.\overline{\alpha} & \overline{\mathbf{t}} = \mathbf{t}
\end{array}
$$

Figure 2.3: Session Types Syntax

structures that are used as a medium for message exchange between processes. This gives a notion of asynchrony to the system. This algebra is studied with *bag* structures, fifo queues and lifo queues. Bags can be considered as a set of messages. A sending process can put a message to a bag and a receiving process can take any message from the bag.

The authors show an encoding between an asynchronous π calculus process to a strongly bisimilar process of their bag structured algebra and the opposite direction on the weak bisimulation. An interesting fact is that it is shown that there is no correspondence between asynchronous π calculus augmented with a lifo structure or a fifo structure.

2.2 Session Types

Session types are used to type for typing π calculus sessions. Communication in a session follows a protocol of exchanging messages. These messages can either be names or values. Session types construct types based on the protocol interaction. If we have consistent session type in each endpoint of the communication that means that we have a communication that can never stuck and will probably terminate. We say probably because in the case of recursive session types we cannot be sure if the communication terminates or not.

The session types constructs that are build on the derivation tree of a π calculus program are useful to have certain properties around soundness. Subject reduction and Type safety are two of these properties. If the typing system preserves soundness then we can catch erroneous programs at compile time.

In this section we will consider the syntax and the typing system of the system described in the previous section. This will give a clear view of session types and what will follow in the rest of the paper.

In figure 2.3 we see the definition and construction of session types. A sort S is structure declaring number, boolean or a name type. Name types are types for π calculus names and it is constructed by session types denoted by α, overline α denotes the dual of a type also defined in this figure. Session types are constructed using prefixes followed by session types. $\mathtt{end}, \bot$ are the basic types and show the end of a session type. A prefix can denote sending a sort or a session type or a label, receiving a sort or a session type or a label. Session type sending is used to model delegation. Recursive types follow the *lambda* calculus definition.

Dual types are also defined in figure 2.3. Dual of the end session is end session. The major concern here is the dual of the prefixes. Sending prefixes have the corresponding receiving prefix as their dual prefix. Dual of recursion considers the dual of the type inside the recursion.

In figure 2.4 we defined the typing system for the algebra defined previously. The typing system runs along the construction tree of the calculus and by an environment it constructs types for names and channels of the program. First the basic types and operators are typed. Then the **inact** process is types with a complete typing. A complete typing is a typing containing only channels typed with the `end` type. (BOT) rule converts end session types to bottom types. The next eight rules describe

$$\Gamma \cdot \alpha : S \vdash \alpha \triangleright S \quad \text{(NAME)} \qquad \frac{\Gamma \vdash e_i \triangleright \mathsf{nat}}{\Gamma \vdash e_1 + e_2 \triangleright \mathsf{nat}} \quad \text{(SUM)}$$

$$\Gamma \vdash 1 \triangleright \mathsf{nat} \quad \text{(NAT)} \qquad \Gamma \vdash \mathrm{tt}, \mathrm{ff} \triangleright \mathsf{bool} \quad \text{(BOOL)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}}{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \bot} \quad \text{(BOT)}$$

$$\frac{\Delta \text{ completed}}{\Theta; \Gamma \vdash \mathit{inact} \triangleright \Delta} \quad \text{(INACT)}$$

$$\frac{\Gamma \vdash \alpha \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash \text{accept } a(k) \text{ in } P \triangleright \Delta} \quad \text{(ACC)} \qquad \frac{\Gamma \vdash \alpha \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \overline{\alpha}}{\Theta; \Gamma \vdash \text{request } a(k) \text{ in } P \triangleright \Delta} \quad \text{(REQ)}$$

$$\frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k![\tilde{e}]; P \triangleright \Delta \cdot k :![\tilde{S}]; \alpha} \quad \text{(SEND)} \qquad \frac{\Theta; \Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k?[\tilde{x}] \text{ in } P \triangleright \Delta \cdot k :?[\tilde{S}]; \alpha} \quad \text{(RCV)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta; \Gamma \vdash \text{throw } k[k']; P \triangleright \Delta \cdot k :![\alpha]; \beta \cdot k' : \alpha} \quad \text{(THR)} \qquad \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot k' : \alpha}{\Theta; \Gamma \vdash \text{catch } k?[x] \text{ in } P \triangleright \Delta \cdot k :?[\alpha]; \beta} \quad \text{(CAT)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha_j \quad j \in I}{\Theta; \Gamma \vdash k \lhd l_j; P \triangleright \Delta \cdot k : \oplus\{l_i : \alpha_i\}_{i \in I}} \quad \text{(SEL)}$$

$$\frac{\forall i . i \in I \quad \Theta; \Gamma \vdash P_i \triangleright \Delta \cdot k : \alpha_i}{\Theta; \Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot k : \&\{l_i : \alpha_i\}_{i \in I}} \quad \text{(BR)}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \quad \Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \quad \text{(IF)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta' \quad \Delta \doteq \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \quad \text{(CONC)}$$

$$\frac{\Theta; \Gamma \cdot \alpha : S \vdash P \triangleright \Delta}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, \alpha)\ P \triangleright \Delta} \quad \text{(NRES)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \bot}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, k)\ P \triangleright \Delta} \quad \text{(CRES)}$$

$$\frac{\Theta \cdot \mathbf{X} : \tilde{S}\tilde{\alpha}; \Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{k} : \tilde{\alpha} \quad \Theta \cdot \mathbf{X} : \tilde{S}\tilde{\alpha}; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \mathtt{def}\ \mathbf{X}\langle \tilde{x}\tilde{k} \rangle = P\ \mathtt{in}\ Q \triangleright \Delta} \quad \text{(DEF)}$$

$$\frac{\Delta \text{ completed} \quad \Gamma \vdash \tilde{e} \triangleright \tilde{S}}{\Theta \cdot \mathbf{X} : \tilde{S}\tilde{\alpha}; \Gamma \vdash \mathbf{X}[\tilde{e}\tilde{k}] \triangleright \Delta \cdot \tilde{k} : \tilde{\alpha}} \quad \text{(VAR)}$$

Figure 2.4: Type System

how session types are build along channel prefixes. Rule (IF) expects both branches to have the same type. The interesting part here comes with rule (CONC). Here the notions of compatibility and composition are used. These two notions are essential into checking the correctness of session interaction. Their definitions are just below. Rules (NRES, CRES) give the typing after restriction and finally the last two rules type variables (VAR) and recursive definitions (DEF).

Definition 2.2.1 *Typings* Δ, Δ *are compatible, written* $\Delta \doteq \Delta$, *if* $\Delta(k) = \overline{\Delta'(k)}, \forall\ k \in dom(\Delta) \cap dom(\Delta')$. *When compatibility holds then the composition, written* $\Delta \circ \Delta'$, *is defined* $\Delta \circ \Delta'(k) = \perp, k \in dom(\Delta) \cap dom(\Delta')$, $\Delta(k), k \in dom(\Delta) \backslash dom(\Delta')$, $\Delta'(k), k \in dom(\Delta') \backslash dom(\Delta)$, *undefined otherwise.*

The above definitions and their usage in the (CONC) rule wants the corresponding endpoints of a session to have dual equality. This gives consistency to the interaction on a channel. If the compatibility criteria hold then on parallel composition the channel is typed with the bottom type and it is essentially eliminated from the typing.

Next the article it presents and proves a subject reduction and a type safety theorem. A more liberal session type system follows which is a different definition for session types systems. Again the whole calculus and typing systems are defined and soundness is proved through subject reduction and type safety.

2.3 Higher Order Asynchronous Session Type Discipline

In [20] we have a definition of a Higher order π calculus formalism. It is basically λ calculus equipped with π calculus constructs under a session type discipline. This work also considers asynchrony in communication.

Asynchronous communication is implemented using queue endpoints. Instead of synchronous communication a process sends a construct to a queue endpoint and receives a construct from another queue endpoint. These endpoints are created when the session is created. This way a sending process does not have to wait for the corresponded process to reach the receiving action to exchange messages. It sends the message to a queue and proceeds with the computation.

This system also describes the possibility of optimizations when sending or receiving messages. The basic idea is that we can permute the sending/receiving prefixes in a process derivation so we can have more data to be communicated and less "'waiting"' time. The permutation can happen if the correctness of the initial specification is not compromised.

This model is essentially a hybrid model for higher order computation and message passing computation. In my opinion it is a very important model that it is closer to the computation done in distributed applications. Of course distributed applications are more based on the object oriented model but the higher order model is a general model as object orientation is.

In this work we are heavily based on asynchronous communication to achieve our goals. Although Higher Order π calculus can be described with synchronous communication, the system in this work cannot exist without asynchronous communication. Also a similar structure to [20] is followed to present the algebra and prove the desired properties, so a good study of this article is recommended.

2.4 Runtime Typing, Exception Handling and the Typecase Consrtuct

Static typing provides safe, sound and error free programs. A sound type system can guaranty that the program can never get stuck at least for the situation the type system checks. It is a very useful tool for the developer as it reveals errors and most often subtle errors at compile time without the need of running the program. But is it the case that statically checked programs can reveal all the possible errors that can happen. The answer is no. There are situations where a static type system

cannot predict the runtime behaviour of a construct just by doing static analysis. Some examples are division by zero, reference to a null object handler or reference an array outside of its limits.

To overcome these problems runtime typing was introduced. This discipline introduces checks during runtime before executing the debated operations and decides whether the execution of the operation can safely take place. In the case where a typing error is present the execution raises an exception. According to the constructs provided by the language and the program specification an exception can be handled with by an exception handler or it can simply terminate the execution and manifest itself through the output. In the case where an exception handler exists then this handler is either developed by the programmer or it is a part of the runtime framework. In either case this is a way to handle situations such as runtime errors. A relative work for exception handling in session types can be found in [7].

This algebra is a session type algebra that gives the possibility of declaring along with a process an exception handler process. The process during its execution can throw an exception on a session. After the exception is thrown the affected parties continue their execution in the exception handler process. This is an idea that tries to embed dynamic type checking in a static typed checking environment such as session types.

Another way to dynamically handle types is the typecase construct. The typecase construct is used when we want to explicitly introduce runtime typing. A discussion about the typecase construct can be found in [1], [2]. These two papers where written fifteen years ago discussing in general runtime type checking in programs that have also been typechecked statically. Indeed several application need cannot determine the type that is handled in an operation at static time event with an advanced typing system. The typecase construct is introduced as a construct for the developer that enables typechecking at runtime and appropriate action according to the recognised type. [1] presents the typecase construct to extend λ calculus. Semantics and soundness is given for this formalism through the paper. [2] again discussed the same issues in a polymorphic environment. Polymorphic programs can be checked statically through supertypes but the evaluation is done at runtime. Explicit or implicit polymorphism can be done using subtyping and the typecase and the dynamic construct. The authors here present their ideas through an ML like system.

Chapter 3

Event Driven Programming

3.1 Asynchrony

Asynchrony plays an important part of a computation in a system. Almost all computations deal with asynchronous events that can happen during the computation and make sense for the desired result. The need from asynchrony comes from our need to have interaction between modules of a system. In almost all systems we allow the interaction of a computation with other computations, with other modules of the system like I/O, users can give information to a computation during the computation and asynchrony can even happen internally in a computation when two logical components communicate in an asynchronous fashion.

Asynchrony deals with data that are produced somewhere else in the system and can be forwarded to the reference computation almost at any time point. Of course the data must have some meaning and their processing should be predicted. Nevertheless it is impossible to have desired result in our need for computation without the correct manipulation of asynchrony.

3.2 Event Driver Model

To cope with asynchrony, programmers have described the event driven programming model. An event can be anything that can happen asynchronously and has meaning for a computation. Each event is received and then processed according to the type of the event and the data that carries with it. Events can be produced almost anywhere in a system. The user, an I/O module, another computation or even the same computation can produce events.

As one can understand from the description above event is a very wide and often subtle notion. There is not a clear description for what is an event and there is no clear description on who produces the events. Generally someone can interpret and use the notion of what is an event according to its needs. But generally there are some characteristics that cannot be changed.

A characteristic that we can conclude is that event driven discipline follows concurrency and parallelism. In order to program event driven applications we must be familiar with the notions and the basics of concurrent and parallel programming. Communication also plays an important role when dealing with this kind of situations. Another characteristic is that of asynchrony. Events can happen at almost any time.

Many libraries, programming languages, OS extensions, compilers etc where developed over the years as tools for developers to handle event driven programming. In these kind of applications in an event was given a programming instance, e.g. define an event object or a programming event primitive. All applications of this category tried to focus on aspects such as performance, efficiency, usability and tried to include a wider set of application to the event definition.

There are two ways to handle events in an application. Either we spawn new threads to handle them or we create a context for each event and we process it along with the other contexts in one computation stream. Many researchers and developers have also experimented with hybrid event models that spawn threads, each one of them responsible to handle multiple event contexts.

If someone decides to take the first approach then the principles of thread or process programming should be applied. Thread creation, shared memory or message passing communication and synchronisation are some of the characteristic of this kind of computation. For a more advanced study of the subject, the thread processing overhead should be taken into account.

The second approach requires for knowledge such as creating and storing an event context. For example if we are waiting for a response from an event then it is best to store it until it the response is ready and proceed with the processing of another event. These contexts should be distinct, in a sense that contexts exist in its own memory boundaries. Scheduling is also important. The application must be able to decide which context to process next. Communication between events can easily be done since the contexts exist on the same memory space. Communication raises issues around synchronisation, but this should not be a big problem since the context switching is done in known time.

A careful consideration of the first approach will reveal the fact that thread-spawning is actually an instance of the second approach. Context switching, scheduling, communication and synchronization are left to be done by the operating system. Hybrid approaches are actually systems with two levels of event processing. This provides more flexibility and fine tuning to the application.

3.3 Event Examples

Next we describe three applications that can naturally be modelled using the event driven model.

3.3.1 User Interface

The simplest example is a user interface. A user can interact with a system through its I/O. Users create events for an application by typing on the keyboard, use the mouse or other input devices. The events interact with the application, which usually has a module designed only to handle input events. The information from the users is processed by this module and the application follows the user's commands.

Almost every application in every day computers follows this approach. A small subset of this application are word processors, spreadsheets, browsers, games, developing tools, shells and everything on the last layer of an application.

3.3.2 Web Servers

Web server applications are responsible of accepting requests from clients and respond by sending files and other kind of data. Notice here that with the term web server we mean almost every server that provides services in a network. HTTP servers, data base servers, mail servers etc. This kind of applications can easily be model as event driven applications. Each new request from a client is a new event. Requests are done asynchronously and their processing is done in a different context.

Except from web servers almost any other network application can be naturally modelled in the event model. Peer to peer models allow interaction and continuous request - respond patterns through a communication session. This adds more asynchrony to an application and event can easily tackle the problems that arise.

3.3.3 Operating Systems

Operating systems are a special category of applications. They are responsible to handle all the resources in a system. These resources respond asynchronously with respect to the entire system. In order to coordinate all these components, an operating system deploys the principles described above. A basic kernel contains the notions of context switching and scheduling.

An operating system can be described using the event model. The only difference is that asynchrony comes from hardware components. The event model exists in an operating system low level approach. The operating system is essentially the first layer of event processing there is. It is the platform that serves all other applications, event driven or not.

The fact that events in an operating system are in low level hardware form give the property to the operating system to be used as an event driven development platform, rather than be modelled as event application itself. Programming begins after the operating system, that's why we do not usually see operating systems described as event driven applications.

3.4 Literature Related With Event Models

Events have always been in the center of contemporary arguments. The research community argues in favor of the event driven discipline or the the thread based discipline for handling concurrency.

The first event driven programming constructs were introduced in Unix systems. The select/poll system call and non blocking inpout output libraries gave the ability to develop real event driven programs.

At that time a presentation in a conference [22] argues that spawning threads to handle concurrency is not such a good idea. It poses a number of reasons why threads is not a good programming discipline and argues that the event model can handle concurrency is better than threads.

Also in [3] there is a discussion for handling the stack of a programming language in order to implement event driven programs. Events and event context is tried to handled in the stack memory in this paper. A technique called stack ripping is introduced here. Stack ripping is used for the implementation of pure event driven frameworks.

One such framework is described in [16] and it is called Tame. Tame is a framework supported by the compiler and by a library that gives the basic operations to the developer to develop event driven applications. The authors argue that Tame can be used in combination with threads. This way a hybrid model can be developed. Another work worth citing is the event driven library described in [8]

Transactional events are studied under the ML framework in [11], [10]. [10] gives an implementation of transactional events through a Haskell library. We can also see a study on a mathematical framework. Transactional events is an idea that comes from the atomicity of transactions in databases. The basic notion is to have atomic events so we can enforce concurrency.

In general for the pure event driven approach as described above we observe that there is no general type checking when it comes to events. This imposes questions about the safety of such models and implementations. With a session type discipline we can enforce a system to be typed consistent, safe and sound.

As event driven programming developed arguments agains it started to show. In [23] the authors argue that event driven programming when it comes to high concurrency servers is not such a good idea. The argument is that threads in a high concurrent environment of a server can indeed improve the situation of concurrency while events can be complicated to develop and their performance is not good.

The conviniency of events along with the performance of threads led to more hybrid approaches in event driven frameworks. A recent work that implements the actors model in Scala can be found in [12]. The actors model can easily be seen as an event driven model. Scala is a Java extension that supports concurrency. This work presents a library that helps the programmer to develop applications in a concurrent portable and event based framework based on the actor framework. The idea of actors is that an event that happens triggers the code of an actor. The work is Scala proposes a portable event driven framework.

Another interesting idea is the idea presented in [17]. This approach develops a haskell framework that allows the developer to write independent thread code to handle different concurrent situations. The event process happens when the framework does an analysis of the thread code and builds an execution with basic points the input/output procedures that are called by the thread code. The analysis gives an execution tree. Based on that tree a scheduler can efficiently schedule each thread. Events here are the input output system calls of the code. Based on the events the execution happens. Operating system threads provide a context for each concurrent entity similar to the context of each event in the event driven approach. The scheduling is not done by the operating system but by a scheduler that has information from the framework compiler. This enhances

even more the efficiency of the model. It is a very interesting, original idea worth exploring even more.

3.5 Staged Even Driven Architecture - SEDA

Staged Even Driven Architecture, SEDA, is described in [24]. It is an architecture for designing and implementing applications that is based on an event driven model. The article begins with a discussion on events and presentation of results of event driven web applications. It then proceeds with a general discussion on events that underlines basic properties of event driven models. Based on that observation the architecture is described. Through a library that implements the basic of SEDA the authors implemented a web server. The rest of the article is devoted to an extended comparison of popular web servers with the SEDA web server through different benchmarks.

Web servers handle multiple requests from clients at any given time. A lot of try was given to have efficient, robust and scalable systems that can handle peaks more than the average in a fairly good time. The term *well - conditioned* is used in the article to describe linear increase of response time with respect to the increase of the load in a system. Unfortunately many web servers and other throughput application do not scale at that degree after a particular point.

The authors of this paper introduced SEDA as an architecture that can solve this problem and give the designing guidelines for well conditioned applications to be implemented. Event applications can be categorized in four categories.

The first category is the simplest implementation there is, that is for each event a new thread is created that is responsible for handling the event. This model though, due to system and resource limitations, cannot provide well condition properties. After a certain point the system thread overhead is increased dramatically resulting to high response times on all events.

The second category is the implementations based on a thread pool. There is a limit of threads that can be created and maintained in an application. Each event is handled by one thread. If there is no thread available then the event waits in a queue for a thread to become free. This approach tries to give low response time on a number of requests, but if the thread pool becomes saturated then there is no fairness between response times, since there are events waiting and events being processed at the same time.

The third approach models events as finite states machines. A scheduler then decides which action from which machine is to be executed next. This is an approach where scheduling and context of events are done by the application and not the operating system. This approach seems to maintain the properties of well condition application, but there are some major drawbacks. Due to the wide spectrum of design choices that are to be made in implementing an application on this kind of model, it is very dificult to find a correct tuning that can give maximum efficiency to the intentions of the application. There is not a general way for implementing different applications. Each application has its own characteristics and these characteristics should be taken into account when designing.

The final category is a way of implementing event driven applications based on event queue constructs in order to make the application more modular. Modularity targets the well condition problem because we are able to tune each module to the systems needs.

After the above discussion SEDA is introduced. SEDA is an architecture that has as a central entity the stage. Stages are the robust building blocks of a SEDA application. Each application is logically decomposed into functional parts. Each part is implemented using a stage. Data move from stage to stage in the way they would move from logical part to a logical part in the application. This way a staged pipeline is formed. Pipeline adds more concurrency to the system. Events here are the messages that move from stage to stage.

A stage is implemented using an event queue, a thread pool, an event handler and a controller. The event queues stores the receiving contexts from other parts of the application or from input operations. The thread pool dequeues events and process them. The event handler is used by the threads to handle the events and it is also responsible for placing the finished events in other stages event queues. The controller gets measurements of the execution variables, like response time and

throughput, and adjusts the resources in a stage. This way the performance of a stage is tuned.

The design parameters of a SEDA application can vary. One parameter is the number of events handled by each thread. Each thread dispatches a number of events from the event queue and uses the event handler to process them. The size of the thread pool also is an important parameter. Resource allocation should consider system specifications. The number of the thread pools is important, meaning that we can have more than one stage sharing a thread pool. An application can be decomposed in several different ways. The numbers and functionality of stages plays an important role in performance.

Some parameters are controlled in runtime by the stage controller. The controller measures at runtime the throughput and the response time of the system or parts of the system. According to that parameters it then adjusts the batching factor and number of threads in the staged, providing fine tuning for performance. In the paper only these two controllers are mentioned, but of course many other controllers can be implemented that use parameters from the entire system in order to have a runtime adjustment of the system.

Under any design choice, SEDA is an architecture that allows us to see an application as a network of functions. Events are the interactions of each module in the application network. By breaking an application into smaller modules we can easily control the performance of each module under the desired specification. Many adjusted modules composed an equally adjusted application. That is the idea behind SEDA. Each stage uses an event driven approach that can be fine tuned according to desire and that provides design flexibility over different application. The rest are left to the programmer, to handle the parameters according to the needs.

The article proceeds by implementing a SEDA web server. The server is builded on a SEDA library that uses an asynchronous input/output layer. The library implements the stage entity and through an interface it can be programmed. The implementation is done on Java. A comparison between the SEDA server called Haboob, the Apache web server and the Flash web server is made.

Some interesting results are that under 256 clients the Haboob server had the highest throughput, followed by Flash and then Apache. Response time was lower for Haboob for both max and average. Flash followed and Apache came last. An interesting result is that the Apache server had average response time close to the other two servers but had a max time almost more than two times bigger than Flash. This is expected because Apache uses a thread pool. The fairness of all three is in the same levels.

When experimenting with 1024 clients the results change. Throughput classification does not change. Apache wins in the average response time but it is worst almost three times from the second in max response time. Haboob comes second in average and first in max response time while Flash is last on average and second in max response time. Fairness is the same between Haboob and Flash but drops significally for Apache.

These results lead us to think that the Haboob server has a more balance and more efficient performance than other servers that follow different design philosophies. The results show a linear increase of parameters with the increase of load. As already discussed if we are not interested for linearity, this can be achieved by tuning parameters in the stage entity. This can be done when we want the application to work under special condition and scalability is not our concern.

SEDA is indeed an event driven architecture that can be used not just for web applications, but almost for every computer application. Event flow in a functionality network can describe many programs in many different fields. An event driven programming model can be used in combination with Staged Event Driven Architecture as a general programming model.

Chapter 4

Session Discipline for Event Driven Programming Model

4.1 Introduction

Event-based programming is a programming model with central entity the event. Events can be anything that "can happen" during a computation. It can be I/O in a more "asynchronous" way or it can be data that were produced by another computation and were received by the current computation. These cases are the most usual cases for events. Generally speaking event with respect to a given computation can be anything that has meaning and can interact with the computation.

The main property of event-based programming is that it requires concurrency to take place. After all events are data that were produced somewhere else and forwarded to the current computation.

Two paths exist in handling events: a multithread approach that spawns threads to handle an event or events and single thread approach that schedules events and handles them as independent entities in the same instruction stream.

No matter what path is selected to implement events the principles of concurrency must be taken into account. Scheduling, synchronization, independence of entities - events, communication and safety are some of the aspects of concern. Concurrency is essential knowledge when it comes to study event driven computation.

Studying events in a theoretical basis should take into account its concurrent nature. Session types are a theoretical tool developed to study concurrency and safety of communication between concurrent computations. As with other concurrent systems, an event based system can be described using session types.

In this work we develop a calculus that can model events in a more theoretical basis always bearing in mind that the notions defined in this algebra are a realistic representation of what exists in implementation. The goal is to keep it simple and can produce simple specifications without the need of implementing complex data structures and other mechanisms.

The approach we decided to follow is the one thread approach for handling the events. Multiple computations do exist but no new processes are spawned to handle an event. An approach such as that requires queues, scheduling and other synchronization mechanisms. There is a try to provide all these as a part of the language definition and semantics

Another important issue that we must address is that of asynchrony. Events can happen at any time of a certain computation. Using the experience through asynchronous session types we try to solve the problem of asynchrony. Again queuing theory is used.

The main construct of session types are sessions. In this algebra we consider sessions as events. More specifically an event is the state of a session. A state of a session is described through the session typing. According to the session type a process can handle the event.

In order to have a complete calculus a series of notion should be defined. Following previews works of session type systems, next in this chapter we describe formally the syntax, terms and terminals of the calculus, the reduction and operational semantics. A static type system is necessary

to catch operational inconsistencies. As we will see a runtime typing system is also needed to later study the properties of this algebra.

4.2 Event Driven Model

Before we proceed with the technical description of the system, a detailed abstract description of the Event Driven Model that is being formalized in this chapter is done. This discussion justifies our choices done for defining the algebra and hopefully it will give the reader a chance for a better understanding of the definition.

We are trying to formalize a model that does not spawn new threads for handling each event. Instead it creates a new context for each event and the processing is done on that context. If needed the context is saved in the memory so it can be recalled on a later stage in order to continue computation on that particular context.

Specifically upon receiving an event a process checks for its type. When this is determined the computation proceeds accordingly. When we are waiting for an event to respond then we can store it along with its context in a data structure and proceed by processing a new event or an already stored context.

This approach is similar with the approach taken by operating systems to handle processes. Each process exists in its own context and can be pre-empted if we are waiting for input/output to occur. So notions such as queues and other data structures and scheduling should be considered in this discipline.

Summarising, the key points we need to take into account are the event checking in order to determine the type of an event, the store of an event context, the scheduling of the events and defining and handling data structure that we might use for storing and retrieving events.

If we consider events to be modelled as session types then a discipline involving types should be developed for session checking. In asynchronous session types endpoint configuration is used for exchanging messages. The properties of these endpoints are ideal for storing event contexts. Following the properties of session endpoints scheduling can be considered as simple round robin policy that emerges from the FIFO policy of the queues. Also we need to define expression construct that give information about endpoint configuration so we can correctly handle them.

A possible event can have many possible types. If we consider modelling events as session types we should allow for session types to carry more than one type at the same time. For this purpose a whole typing system along with proper sub-typing relation should be defined.

4.3 Syntax

Next we give the syntax of our system. Terminals, terms and syntax construction are given. All the symbols concerning syntax that are being used through the work and adopted by convention are given in this section

4.3.1 Terminals and Identifiers

In figure 4.1 the definition of terminals is given. We consider that shared channels range over $a, b, c, \ldots$, session channels range over $s, s', \ldots$ and labels range over $l_1, l_2, \ldots$. Boolean constants true, false are denoted by the symbols $\mathtt{tt}, \mathtt{ff}$ respectively. Natural number constants are also being used.

We define four types of variables. Variables range over $x, y, z, \ldots$, process variables range over $X, Y, Z, \ldots$, type variables range over $\mathbf{t}, \mathbf{t}', \ldots$ and match variables range over $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$.

It is useful that symbols $u, u', \ldots$ can be used to denoted either variables or shared channels. Similarly symbols $k, k', \ldots$ can be either variables or session channels.

Finally values $v, v', \ldots$ can be natural numbers, boolean, shared channels or session channels.

Notice that the primitive constant types are boolean and natural number. This algebra can also be extended with other primitive types.

Shared Channels	$a, b, c, \ldots$
Session Channels	$s, s', \ldots$
Labels	$l_1, l_2, \ldots$
Boolean Constants	tt, ff
Number Constants	$0, 1, 2, \ldots$
Variables	$x, y, z, \ldots$
Process Variables	$X, Y, Z, \ldots$
Type Variables	$\mathbf{t}, \mathbf{t}', \ldots$
Match Variables	$\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$

$$u, u' \quad ::= \quad x, y, z \mid a, b, c$$
$$k, k' \quad ::= \quad x, y, z \mid s, s'$$

$$v, v' \quad ::= \quad 0, 1, 2, \ldots \mid \mathrm{tt}, \mathrm{ff} \mid a, b, c \mid s, s'$$

Figure 4.1: Terminal symbols

4.3.2 Expressions

Expressions are needed so we can express computation through our system. Operators that express computation between values are defined here. Some operators such as `arrived` k and `arrived` a are essential to the definition and meaning of the whole algebra.

Expression range over $e, e', \ldots$ and can be either values, variables, boolean operators, arithmetic operators and the operators `arrived` k and `arrived` a.

$$e, e' \quad ::= \quad v, v' \mid x, y, z \mid \texttt{not}(e), \ \ldots \mid e + e, \ \ldots$$
$$\texttt{arrived}\ k \mid \texttt{arrived}\ u$$

4.3.3 Terms and syntax

In figure 4.2 we have the syntax structure of the algebra. Session initiation happens with (Session Accept) and (Session Request) constructs. These constructs create new sessions for communication between processes. (Output) and (Input) represent output and input session communication respectively. (Select) and (Branch) are used for select and branching. The (Conditional) construct is the standard control construct. Queues (Session Channel Endpoint Configuration) are used to describe asynchronous communication. (Shared Channel Endpoint Configuration) are used to describe asynchronous session initiation. The (Typecase) construct is used for event matching and processing. Recursion is defined using the (Agent Definition Scope) and (Agent Instance). A process can be defined and recursively instantiated using these two constructs. To complete the algebra (Parallel Composition), (Shared Channel Restriction), (Session Channel Restriction) and the null process (Completed Process) are defined.

Values that are contained in a session queue are denoted by h and can be either values v or labels l. Shared channel queues can contain only sessions s. Symbols D, D' are defined in (Agent Definition). This term is used to define recursion.

The contributions of this algebra are the constructs used to describe and implement events. The (Typecase) is used to match a session channel with a session type and proceed to computation accordingly. In order to match types a type discipline should be defined later. This is on of the key construct to this algebra and it is the one that makes this algebra differ from other session

$$
\begin{array}{lcll}
P, Q & ::= & u(x : S).P & \text{(Session Accept)} \\
& | & \overline{u}(x : S).P & \text{(Session Request)} \\
& | & k\,!\langle e\rangle; P & \text{(Output)} \\
& | & k\,?(x); P & \text{(Input)} \\
& | & k \triangleleft l; P & \text{(Select)} \\
& | & k \triangleright \{l_i : P_i\}_{i \in I} & \text{(Branch)} \\
& | & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q & \text{(Conditional)} \\
& | & P \mid Q & \text{(Parallel Composition)} \\
& | & (\boldsymbol{\nu}\ a)\ P & \text{(Shared Channel Restriction)} \\
& | & (\boldsymbol{\nu}\ s)\ P & \text{(Session Channel Restriction)} \\
& | & \texttt{def}\ D\ \texttt{in}\ P & \text{(Agent Definition Scope)} \\
& | & \mathbf{X}\langle \tilde{e} \rangle & \text{(Agent Instance)} \\
& | & \mathbf{0} & \text{(Completed Process)} \\
& | & a\,[\langle S\rangle] : \vec{s} & \text{(Shared Channel Endpoint Configuration)} \\
& | & s\,[S] : \vec{h} & \text{(Session Endpoint Configuration)} \\
& | & \texttt{typecase}\ k\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)S_i : P_i\}_{i \in I} & \text{(Typecase)}
\end{array}
$$

$$
h \quad ::= \quad v \quad | \quad l \qquad \text{(Buffered Message)}
$$

$$
D, D' \quad ::= \quad \mathbf{X}_1\langle \tilde{x}_1 \rangle = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ \mathbf{X}_n\langle \tilde{x}_n \rangle = P_n \qquad \text{(Agent Definition)}
$$

Figure 4.2: Process Syntax

type systems. Matching is an essential concept of the event model that we are trying to capture throught this discussion.

Another novelty of this algebra is asynchronous session initiation. This is done using shared channel queues (Shared Channel Endpoint Configuration). This is useful as we can see later because we can have a session queue structure not only as communication endpoint between processes but as an event store medium without implementing other complex data structures.

4.3.4 Substitution

Substitution of most construct is standard as defined in the literature, but we need to define substitution for the new constructs in the algebra.

The (Typecase) term substitution substitutes the expression term and all processes defined in the body of the construct. The (Session Channel Endpoint Configuration) and (Shared Channel Endpoint Configuration) queue substitution performs the substitution in every element of the queue.

$$
\begin{array}{l}
(\texttt{typecase}\ e\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)S_i : P_i\}_{i \in I})\,\sigma \triangleq \texttt{typecase}\ (e\sigma)\ \texttt{of}\ \{(\mathsf{X}_i)S_i : (P_i)\sigma\}_{i \in I} \\
(a\,[\langle S\rangle] : s_1, \ldots, s_n)\,\sigma \triangleq a\,[\langle S\rangle] : s_1\sigma, \ldots, s_n\sigma \\
(s\,[S] : h_1, \ldots, h_n)\,\sigma \triangleq s\,[S] : h_1\sigma, \ldots, h_n\sigma
\end{array}
$$

4.3.5 Structural Congruence

Structural congruence is given in figure 4.3. As we can see we have a standard definition of structural congruence following the literature.

$$
\begin{array}{rcll}
P & \equiv & Q & \text{if } P \equiv_\alpha Q \\
P \mid \mathbf{0} & \equiv & P & \text{(Idempotence)} \\
P \mid Q & \equiv & Q \mid P & \text{(Commutativity)} \\
(P \mid P') \mid P'' & \equiv & P \mid (P' \mid P'') & \text{(Associativity)} \\
 & & & \\
(\boldsymbol{\nu}\, a : \langle S \rangle)\, \mathbf{0} & \equiv & \mathbf{0} & \text{(Shared Channels)} \\
(\boldsymbol{\nu}\, a : \langle S \rangle)\, P \mid Q & \equiv & (\boldsymbol{\nu}\, a : \langle S \rangle)\, (P \mid Q) & \text{if } a \notin \mathrm{fn}(Q) \\
(\boldsymbol{\nu}\, a : \langle S \rangle)\, \mathtt{def}\ D\ \mathtt{in}\ P & \equiv & \mathtt{def}\ D\ \mathtt{in}\ (\boldsymbol{\nu}\, a : \langle S \rangle)\, P & \text{if } a \notin \mathrm{fn}(D) \\
 & & & \\
(\boldsymbol{\nu}\, s)\, \mathbf{0} & \equiv & \mathbf{0} & \text{(Session Channels)} \\
(\boldsymbol{\nu}\, s)\, P \mid Q & \equiv & (\boldsymbol{\nu}\, s)\, (P \mid Q) & \text{if } s \notin \mathrm{fs}(Q) \\
(\boldsymbol{\nu}\, s)\, \mathtt{def}\ D\ \mathtt{in}\ P & \equiv & \mathtt{def}\ D\ \mathtt{in}\ (\boldsymbol{\nu}\, s)\, P & \text{if } s \notin \mathrm{fs}(D) \\
 & & & \\
\mathtt{def}\ D\ \mathtt{in}\ \mathbf{0} & \equiv & \mathbf{0} & \text{(Agent Definition Scopes)} \\
(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q & \equiv & \mathtt{def}\ D\ \mathtt{in}\ P \mid Q & \text{if } \mathrm{dpv}(D) \cap \mathrm{fpv}(Q) = \emptyset \\
\mathtt{def}\ D\ \mathtt{in}\ (\mathtt{def}\ D'\ \mathtt{in}\ Q) \mid Q & \equiv & \mathtt{def}\ D\ \mathtt{and}\ D'\ \mathtt{in}\ P & \text{if } \mathrm{dpv}(D) \cap \mathrm{dpv}(D') = \emptyset
\end{array}
$$

Figure 4.3: Structural Congruence

4.4 Type Syntax

Before we proceed we need to define notions around typing and session typing. These definitions are needed in order to give reduction, operational semantics and type system to the algebra.

4.4.1 Session types

In figure 4.4 the session type and type discipline is defined. Symbol U denotes basic types such as boolean type, natural type, label type, shared channel type, top and bottom types. Symbol T is used either for basic types or Session types.

This typing system is designed to support subtyping and a notion of matching. Subtyping and matching are defined later in this section.

Session types are constructed using output, input, select and branch constructs. Basic constructs consists of the *end* defining the inactive session type. Recursive session types are defined using the λ calculus notation for recursive types. A contribution to the algebra is the definition of Session Set Types and match variables. Session set types consider a set of session types as a session type. Match variables are used combined with the matching discipline as defined later in this section. The top and bottom elements are used to complete the subtyping lattice.

We also need to define equivalence laws for session set types. A set type that has as an element a set type is equal to the element and a set type that has many elements as set types is a set type with the union of all session types in the set type elements.

The introduction of set session types plays an important role as far as subtyping as presented below. The combination of types in set session types provides creates mutual subtypes for the combine types. This offers great static flexibility when dealing with events whose type is unknown at compile time.

4.4.2 Type Duality

The following laws give define the dual of types. Dual types are used in the typing system to check for communication correctness.

$$
\begin{array}{lcll}
S & ::= & !\langle T\rangle; S & \text{Session Output} \\
 & | & ?(T); S & \text{Session Input} \\
 & | & \oplus\{l_i : S_i\}_{i\in I} & \text{Session Selection} \\
 & | & \&\{l_i : S_i\}_{i\in I} & \text{Session Branch} \\
 & | & \mu \mathbf{t}.S & \text{Session Recursion} \\
 & | & t & \text{Recursion Variable} \\
 & | & \mathtt{end} & \text{Session end} \\
 & | & \mathsf{X}, \mathsf{Y}, \mathsf{Z} & \text{Match Variables} \\
 & | & [S_i]_{i\in I} & \text{Session set types} \\
 & | & \top & \text{Top} \\
 & | & \bot & \text{Bottom} \\
 & & & \\
T & ::= & U \mid S & \\
U & ::= & \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{lab} \mid \langle S\rangle \mid \top \mid \bot & \\
 & & & \\
 & & [S] = S & \\
 & & [[S_i]_{i\in I_1}, [S_i]_{i\in I_2}] = [S_i]_{i\in I_1\cup I_2} &
\end{array}
$$

Figure 4.4: Session Types

$$
\begin{array}{ll}
\overline{!\langle T\rangle; S} = ?(T); \overline{S} & \overline{?(T); S} = !\langle T\rangle; \overline{S} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} = \&\{l_1 : \overline{S}_1, \ldots, l_n : \overline{S}_n\} & \overline{\&\{l_i : S_i\}_{i\in I}} = \oplus\{l_1 : \overline{S}_1, \ldots, l_n : \overline{S}_n\} \\
\overline{\mu \mathbf{t}.S} = \mu \mathbf{t}.\overline{S} & \overline{\mathbf{t}} = \mathbf{t} \\
\overline{\mathtt{end}} = \mathtt{end} & \overline{\overline{S}} = S
\end{array}
$$

4.4.3 Subtyping

In this system we find it convenient to define subtyping. Our approach to the event model is heavily relied on session set types and matching. Session set types can be understood if a proper subtyping relation is given. Matching definition takes into account the subtyping relation. The subtyping relation is also used in the typing system.

Figure 4.5 gives the subtyping relation. This relation is a binary relation between types and is in fact a lattice on the space of all types, session types and basic types that are defined in figure 4.4. To calculate a possible subtyping between two types we need to calculate their greater fix point in this lattice. We write $S \leq S'$ for $(S, S') \in \mathbf{R}$ and is read S is a subtype of S'.

4.4.4 Type Matching

In figure 4.6 we define the rules that match two session types. The matching of session types is used as an operation in the runtime of our system in order to recognize and later on process events. As described events are and can be thought as session types. The whole concept of matching becomes much clearer when the operational semantics are given.

Matching is checked and performed on the derivation tree of a session type. Match variables are matched against session types and are substituted through the derivation tree in order to come up with a correct matching.

We can see two axioms. A type is matched with itself and a type is matched with a matching variable and also gives a substitution. Derivation on the session types being matched is performed by the following rules, describing output, input, selection, branching and recursion.

$$
\begin{array}{rcl}
\mathbf{R} & = & \{(\top, \top)\} \\
& \cup & \{(\bot, \bot)\} \\
& \cup & \{T, \top\} \\
& \cup & \{\bot, T\} \\
& \cup & \{(\mathsf{bool}, \mathsf{bool})\} \\
& \cup & \{(\mathsf{nat}, \mathsf{nat})\} \\
& \cup & \{(\mathsf{lab}, \mathsf{lab})\} \\
& \cup & \{(\langle S \rangle, \langle S' \rangle) \mid (S, S') \in \mathbf{R} \wedge (S', S) \in \mathbf{R}\} \\
& \cup & \{(!\langle T \rangle; S, !\langle T' \rangle; S') \mid (T, T'), (S, S') \in \mathbf{R}\} \\
& \cup & \{(?(T); S, ?(T'); S') \mid (T', T), (S, S') \in \mathbf{R}\} \\
& \cup & \{(\oplus\{l_i : S_i\}_{i \in I}, \oplus\{l'_i : S'_i\}_{i \in I'}) \mid \{l_i : S_i\}_{i \in I} \subseteq \{l'_i : S'_i\}_{i \in I'}, \forall i \in I, (S_i, S'_i) \in \mathbf{R}\} \\
& \cup & \{(\&\{l_i : S_i\}_{i \in I}, \&\{l'_i : S'_i\}_{i \in I'}) \mid \{l'_i : S'_i\}_{i \in I'} \subseteq \{l_i : S_i\}_{i \in I}, \forall i \in I', (S_i, S'_i) \in \mathbf{R}\} \\
& \cup & \{(\mu \mathbf{t}.S, S') \mid (S\{\mu \mathbf{t}.S/t\}, S') \in \mathbf{R}\} \\
& \cup & \{(S, \mu \mathbf{t}.S') \mid (S, S'\{\mu \mathbf{t}.S'/\mathbf{t}\}) \in \mathbf{R}\} \\
& \cup & \{(\mathtt{end}, \mathtt{end})\} \\
\\
& \cup & \{([S_i]_{i \in I}, [S'_i]_{i \in I'}) \mid \forall i.i \in I. \exists j.j \in I'.(S_i, S'_j) \in \mathbf{R}\}
\end{array}
$$

Figure 4.5: Subtyping relation

$$
\frac{}{\mathsf{match}(T, T) = \emptyset}
$$

$$
\frac{}{\mathsf{match}(T, \mathsf{X}) = \{T/\mathsf{X}\}}
$$

$$
\frac{\mathsf{match}(T, T') = \sigma \quad \mathsf{match}(S, S'\sigma) = \sigma'}{\mathsf{match}(!\langle T \rangle; S, !\langle T' \rangle; S') = \sigma \cup \sigma'}
$$

$$
\frac{\mathsf{match}(T, T') = \sigma \quad \mathsf{match}(S, S'\sigma) = \sigma'}{\mathsf{match}(?(T); S, ?(T'); S') = \sigma \cup \sigma'}
$$

$$
\frac{\forall i. 1 \leq i \leq n \quad \mathsf{match}(S_i, S'_i) = \sigma_i}{\mathsf{match}(\oplus\{l_1 : S_1, \ldots, l_n : S_n\}, \oplus\{l_1 : S'_1, \ldots, l_n : S'_n\}) = \sigma_1 \cup \cdots \cup \sigma_n}
$$

$$
\frac{\forall i. 1 \leq i \leq n \quad \mathsf{match}(S_i, S'_i) = \sigma_i}{\mathsf{match}(\&\{l_1 : S_1, \ldots, l_n : S_n\}, \&\{l_1 : S'_1, \ldots, l_n : S'_n\}) = \sigma_1 \cup \cdots \cup \sigma_n}
$$

$$
\frac{\mathsf{match}(S, S') = \sigma}{\mathsf{match}(\mu \mathbf{t}.S, \mu \mathbf{t}.S') = \sigma}
$$

Figure 4.6: Matching Rules

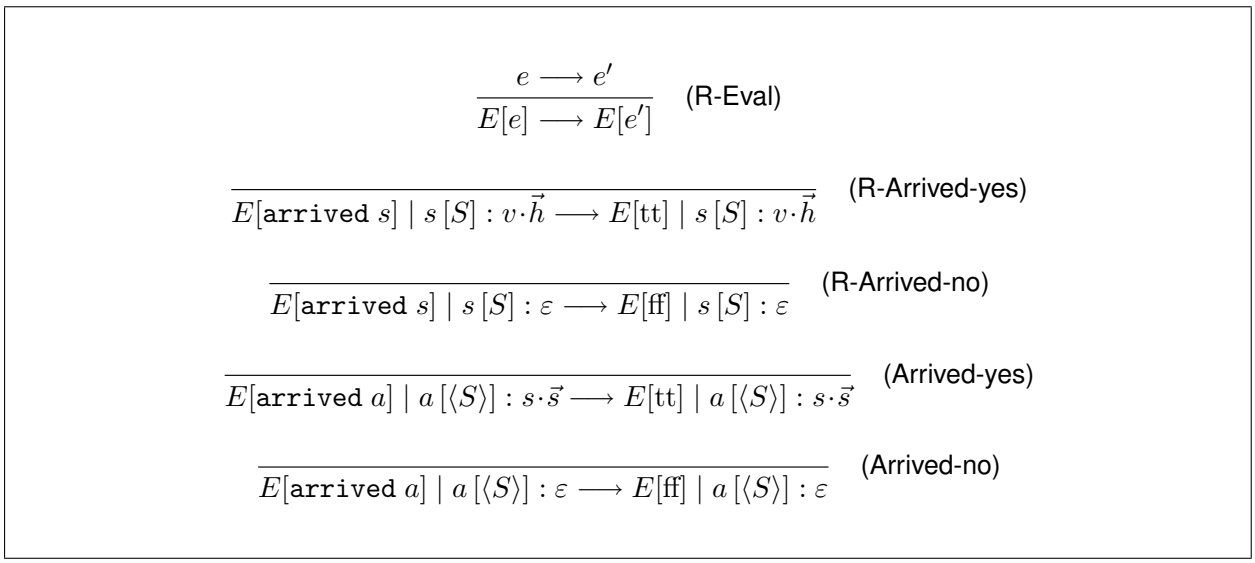

Figure 4.7: Expression Reduction

4.5 Operational Semantics

After defining process syntax and type discipline we need to define operational semantics. The reduction relation of the algebra is defined again in a way that serves the asynchronous communication and event driven programming model.

4.5.1 Expression Contexts

Expression contexts E are used for expression substitution in a process. It replaces all holes $-$ in E with the expression e. $E[e] = E\{e/-\}$. This notion is used to accurately define expression reduction rules. The structure of E is defined as follows.

$$
\begin{array}{lcl}
E' & ::= & - \\
 & \mid & e \\
 & & \\
E & ::= & s\,!\langle E'\rangle; P \\
 & \mid & \texttt{if}\ E'\ \texttt{then}\ P\ \texttt{else}\ Q \\
 & \mid & P \mid Q \\
 & \mid & (\boldsymbol{\nu}\ a : \langle S\rangle)\ P \\
 & \mid & (\boldsymbol{\nu}\ s)\ P \\
 & \mid & \texttt{def}\ D\ \texttt{in}\ P \\
 & \mid & \mathbf{X}\langle \tilde{v} E' \tilde{e}\rangle \\
 & \mid & \texttt{typecase}\ E'\ \texttt{of}\ \{(\vec{\mathsf{X}}_i)S_i : P_i\}_{i\in I}
\end{array}
$$

A basic expression context is either a hole $-$ or an expression. The structure of an expression context E follows the structure of a process. Whenever in the definition of a process we expect an expression a hole can be placed. The reason queues are not appeared in the definition of expression context is because queues can contain only values and not expressions.

4.5.2 Expression Reduction

In figure 4.7 we consider the reduction and computation of expression that exist in an expression context. This includes arithmetic and boolean operators as well as the `arrived` k, `arrived` a operators. These operators are used to check whether an endpoint, shared or session, is empty or not. The event model we present here needs these operators in order to perform computation consistently.

The first rule shows evaluation of an expression. If an expression can be evaluated according to the operators forming it then the expression is evaluated in the expression context. The last four rules are used to describe the reduction of the `arrived` construct. We have two similar pairs, one for each type of queues. The idea is that the expression is evaluated to ff if the checked queue is empty and to tt if the checked queue is not empty.

4.5.3 Operational Semantics

The reduction semantics of the algebra are in figure 4.8. Session initiation is asynchronous and described by rules (Init-req) and (Init-acc). Session endpoints are created dualy one with each queue. Send and receive is described by rules (R-Send) and (R-Reiceve). Send and receive actions interact with an endpoint to send or receive messages. Similarly (R-Select) and (R-Branch) define method call and branching. The (R-If-true) and (R-If-false) describe the control flow according to the boolean evaluation of the if expression.

Parallel composition, channel restriction, congruence, instance and definition scope follow the classic approach of process algebras and π - calculus.

The (R-Typecase-s) rule uses the matching rules to match session types and control the process continuation according to the session type of the matching channel.

We must stress the importance of rules (Init-req) and (Init-acc). These two rules provide an asynchronous way in creating a new session. An asynchronous creation of a session may lead to the creation of a session that is private to only one process. This can be useful because as we demonstrate through examples session endpoints data structures have properties especially useful when handling events in this event driven model. It can be used instead of other complex data structures for storing event types and computation context. The reason for asynchrony is that we need the request and accept action to be performed by one process and its continuation. This way the resulting endpoints are private to a process that is not a parallel composition without using delegation.

We can see the asynchronous session communication is described through rules (R-Send) and (R-Receive). Using the session queues data are stored and retrieved using these two rules.

The choice of an event and the choice to process it comes through the (R-Typecase-s). When receiving an event a process needs to know its type so it can proceed by processing it. In the typecase rule the matching discipline is used to find the appropriate event/session that matches with the event/session being checked. After matching the typecase process proceeds to the process defined after the matched event session in the typecase definition.

To demonstrate the new ideas of these semantics consider as an example the process:

$$\overline{a}(s).s\,!\langle 1\rangle;\mathbf{0} \mid \overline{a}(s).s\,?(x);\mathbf{0} \mid a\,[\langle S\rangle] : \varepsilon \mid$$
$$a(x).\texttt{typecase}\ x\ \texttt{of}\ \{?(\mathsf{nat});\mathsf{end} : x\,?(y);\mathbf{0}, !\langle\mathsf{bool}\rangle;\mathsf{end} : x\,!\langle \mathrm{tt}\rangle;\mathbf{0}\}$$

In this process we can initially observe two actions. One of the first two parallel processes can request a session initiation. Whichever starts a request the session initiation is done asynchronously:

$$\rightarrow$$
$$(\boldsymbol{\nu}\ s)\ (\overline{s}\,!\langle 1\rangle;\mathbf{0} \mid s\,[!\langle\mathsf{nat}\rangle;\mathsf{end}] : \varepsilon) \mid \overline{a}(s).s\,?(x);\mathbf{0} \mid a\,[\langle S\rangle] : s \mid$$
$$a(x).\texttt{typecase}\ x\ \texttt{of}\ \{?(\mathsf{nat});\mathsf{end} : x\,?(y);\mathbf{0}, !\langle\mathsf{bool}\rangle;\mathsf{end} : x\,!\langle \mathrm{tt}\rangle;\mathbf{0}\}$$
$$\rightarrow$$
$$(\boldsymbol{\nu}\ s)\ (\overline{s}\,!\langle 1\rangle;\mathbf{0} \mid s\,[!\langle\mathsf{nat}\rangle;\mathsf{end}] : \varepsilon \mid$$
$$\texttt{typecase}\ s\ \texttt{of}\ \{?(\mathsf{nat});\mathsf{end} : s\,?(y);\mathbf{0}, !\langle\mathsf{bool}\rangle;\mathsf{end} : s\,!\langle \mathrm{tt}\rangle;\mathbf{0}\} \mid \overline{s}\,[?(\mathsf{nat});\mathsf{end}] : \varepsilon) \mid$$
$$\overline{a}(s).s\,?(x);\mathbf{0} \mid a\,[\langle S\rangle] : \varepsilon$$

By actions (Init-req) and (Init-acc) the session has been established. The typecase rule is used to check the type of the session/event that was initiated:

$$\rightarrow$$
$$(\boldsymbol{\nu}\ s)\ (\overline{s}\,!\langle 1\rangle;\mathbf{0} \mid s\,[!\langle\mathsf{nat}\rangle;\mathsf{end}] : \varepsilon \mid s\,?(y);\mathbf{0} \mid \overline{s}\,[?(\mathsf{nat});\mathsf{end}] : \varepsilon) \mid$$
$$\overline{a}(s).s\,?(x);\mathbf{0} \mid a\,[\langle S\rangle] : \varepsilon$$

$$\frac{}{\overline{a}(s:S).P \mid a\,[\langle S\rangle] : \vec{s} \longrightarrow (\boldsymbol{\nu}\; s)\;(P \mid a\,[\langle S\rangle] : \vec{s}{\cdot}s \mid s\,[S] : \varepsilon)} \quad \text{(Init-req)}$$

$$\frac{}{a(x:S).P \mid a\,[\langle S\rangle] : s{\cdot}\vec{s} \longrightarrow P\{x/\overline{s}\} \mid a\,[\langle S\rangle] : \vec{s} \mid \overline{s}\,[\overline{S}] : \varepsilon} \quad \text{(Init-acc)}$$

$$\frac{}{s\,!\langle v\rangle; P \mid s\,[!\langle T\rangle; S] : \vec{h} \mid \overline{s}\,[S'] : \vec{h'} \longrightarrow P \mid s\,[S] : \vec{h} \mid \overline{s}\,[S'] : \vec{h'}{\cdot}v} \quad \text{(R-Send)}$$

$$\frac{}{s\,?(x); P \mid s\,[?(T); S] : v{\cdot}\vec{h} \longrightarrow P\{v/x\} \mid s\,[S] : \vec{h}} \quad \text{(R-Receive)}$$

$$\frac{i \in I}{s \triangleleft l_i; P \mid s\,[\oplus\{l_i : S_i\}_{i\in I}] : \vec{h} \mid \overline{s}\,[S'] : \vec{h'} \longrightarrow P \mid s\,[S_i] : \vec{h} \mid \overline{s}\,[S'] : \vec{h'}{\cdot}l_i} \quad \text{(R-Select)}$$

$$\frac{1 \le i \le \mathtt{min}(m,n)}{s \triangleright \{l_1 : P_1, \ldots, l_m : P_m\} \mid s\,[\&\{l_1 : S_1, \ldots, l_n : S_n\}] : l_i{\cdot}\vec{h} \longrightarrow P_i \mid s\,[S_i] : \vec{h}} \quad \text{(R-Branch)}$$

$$\frac{}{\mathtt{if}\ \mathrm{tt}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P} \quad \text{(R-If-true)}$$

$$\frac{}{\mathtt{if}\ \mathrm{ff}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q} \quad \text{(R-If-false)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \text{(R-Par)}$$

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu}\; a : \langle S\rangle)\; P \longrightarrow (\boldsymbol{\nu}\; a : \langle S\rangle)\; P'} \quad \text{(R-Restr-a)}$$

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu}\; s)\; P \longrightarrow (\boldsymbol{\nu}\; s)\; P'} \quad \text{(Restr-s)}$$

$$\frac{\mathbf{X}\langle\tilde{x}\rangle = P \in D}{\mathtt{def}\ D\ \mathtt{in}\ (\mathbf{X}\langle\tilde{v}\rangle \mid Q) \longrightarrow \mathtt{def}\ D\ \mathtt{in}\ P\{\tilde{v}/\tilde{x}\} \mid Q} \quad \text{(R-Instance)}$$

$$\frac{P \longrightarrow P'}{\mathtt{def}\ D\ \mathtt{in}\ P \longrightarrow \mathtt{def}\ D\ \mathtt{in}\ P'} \quad \text{(R-Def-scope)}$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} \quad \text{(R-Struct)}$$

$$\frac{\exists i.i \in I \quad \mathsf{match}(S', S_i) = \sigma \quad S' \le S}{\mathtt{typecase}\ s : S\ \mathtt{of}\ \{(\vec{\mathsf{X}})S_i : P_i\}_{i\in I} \mid s\,[S] : \vec{h} \longrightarrow P_i \mid s\,[S] : \vec{h}} \quad \text{(R-Typecase-s)}$$

Figure 4.8: Operational Semantics

All that remains is the asynchronous communication. Rules (R-Send), (R-Receive) describe asynchronous communication:

$$
\begin{array}{l}
\rightarrow \\
(\boldsymbol{\nu}\ s)\ (s\,[\mathtt{end}] : \varepsilon \mid s\,?(y); \mathbf{0} \mid \overline{s}\,[?(\mathsf{nat}); \mathtt{end}] : 1) \mid \\
\overline{a}(s).s\,?(x); \mathbf{0} \mid a\,[\langle S \rangle] : \varepsilon \\
\rightarrow \\
(\boldsymbol{\nu}\ s)\ (\mathbf{0} \mid s\,[\mathtt{end}] : \varepsilon \mid \overline{s}\,[\mathtt{end}] : \varepsilon) \mid \\
\overline{a}(s).s\,?(x); \mathbf{0} \mid a\,[\langle S \rangle] : \varepsilon
\end{array}
$$

4.6 Type System

To complete the definition of the algebra a type system is given. This type system is based on the session type philosophy. The fact that the typing system has structures that allow asynchronous communication leads to a separation of the type system rules to static type system and runtime type system. The static type system is used to statically analyze the definition of a process. The runtime typing system is an extension of the static runtime system and it is used to analyse every derivative of an initial process. Runtime typing system typechecks the endpoint structures. To completely typecheck a system we use the whole type system that consisted from the static and runtime type system.

4.6.1 Static Typing System

We begin by introducing basic definitions used through the type discipline.
Expression typing judgements for "constant types" have the form

$$\Gamma \vdash e \triangleright U$$

and process typing judgements are of the form

$$\Theta; \Gamma \vdash P \triangleright \Sigma$$

where

- Γ maps variables (v, v') to constant types (U)
- Θ maps process variables (X, Y, Z) to sequences of types $(\tilde{T})$
- Σ maps session channels (k, k') to session types (S)

Γ and Θ consist the typing environment mappings while Σ is a mapping set with all useful information for a correct typing.

Typing Σ is *completed* iff $\forall k.k \in \mathrm{dom}(\Sigma) : \Sigma(k) = \mathtt{end}$ or $\perp$

We say that typing Σ' is a subtype of typing Σ, denoted $\Sigma' \leq \Sigma$, when $\mathrm{dom}(\Sigma') = \mathrm{dom}(\Sigma), \forall s \in \mathrm{dom}(\Sigma), \Sigma'(s) \leq \Sigma'(s)$

In figure 4.9 we can see all typing rules. Rule (T-Sub-u) is standard rule for subtypes stating that if an expression has also type a supertype of its type. Rule (T-Var-u) states that a variable has type U while rule (T-Shared-chan) defines types for shared channel as $\langle S \rangle$. Next there are rules that type different expressions and constants that are encountered in the language such as (T-Bool-true), (T-Bool-false), (R-Nat), (T-Session-Arrived), (T-Shared-Arrived). The last rule is a contribution of this algebra and types the expression for checking a shared queue channel.

Continuing we have standard rules for session types. (T-Accept) and (T-Request) define the type of a shared channel with respect to the created session channel and insist that the shared channel type is conformant to the environment. Rules (T-Send-u) and (T-Receive-u) construct the basic send

- receive types of a session while rules (T-Send-s) and (T-Receive-s) type the session type delegation. (T-Select) and (T-Branch) give typing for selection/branching constructs. (T-if) types the if control structure where we expect a boolean expression and the two possible continuations to have the same type. In rule (T-Par) parallel composition is checked along parallel processes. The resulting typing is a union of the two parallel processes. Restriction typing is seen in (T-Restr-a). Recursion is described with (T-Def-in) rule and completed with (T-Process-var) rule which gives the typing when process variables are used. The empty process is typed via (T-Nill) rule, that wants the typing set to be complete. This rule is the basic rule that types processes.

Finally the type system extension considers the (R-Typecase-s) construct that is typed using the (T-Typecase) rule. This rule uses set types to type the channel that is being matched in the (R-Typecase-s) rule. Set types complete the construction of session types in this algebra and from this rule we see why we defined set types. When an event occurs we are dealing with many different types. Many types can be expressed by using set session types.

4.6.2 Runtime Typing System

To complete the typing system we need to define a runtime typing system. The following definitions and rules are used to type the runtime constructs, which are session and shared endpoints. Again the typing is done with respect to the given environment and there exists a set that contains all the information useful for consistent typing. There are also definitions and relations that check for typing correctness. Following we have the basic definitions.
Runtime message typing judgements (for enqueued messages) have the form:

$$\Theta; \Gamma; \Delta \vdash v : T$$

and runtime processes have typing judgements of the form

$$\Theta; \Gamma \vdash P \triangleright \Delta$$

where Δ is

$$\Delta \quad ::= \quad \Sigma \quad | \quad \Delta \cdot s : [S]\,\vec{\tau} \quad | \quad \Delta \cdot s : (S, [S']\,\vec{\tau}) \quad | \quad \Delta \cdot a$$

with

$$\tau \quad ::= \quad T \quad | \quad l$$

Each process has a runtime typing. Runtime typing is composed first by getting the result of the static typing and then we have a mapping from a session endpoint containing the Session type of the endpoint and the values in the queue. Also we can find the construct $s : (S, \vec{\tau})$ and the name of a shared endpoint configuration. The second endpoint mapping differs from the first in a sense that the second emerge from the composition of two runtime sets defined later and it is used for checking the duality of sessions. The first session endpoint mapping is used by a runtime typing rule to type a session endpoint.
The composition of Δ environments $(\Delta \odot \Delta')$ is defined whenever the following hold:

$$(\Delta \odot \Delta')(a) = \begin{cases} \Delta(a) & \text{if } a \in \text{dom}(\Delta) \text{ and } a \notin \text{dom}(\Delta') \\ \Delta(a) & \text{if } a \in \text{dom}(\Delta') \text{ and } a \notin \text{dom}(\Delta) \\ undefined & \text{otherwise} \end{cases}$$

and

$$\frac{\Gamma \vdash e \triangleright U' \quad U' \leq U}{\Gamma \vdash e \triangleright U} \quad \text{(T-Sub-u)}$$

$$\frac{}{\Gamma \cdot x : U \vdash x \triangleright U} \quad \text{(T-Var-u)} \qquad \frac{}{\Gamma \cdot u : \langle S \rangle \vdash u \triangleright \langle S \rangle} \quad \text{(T-Shared-chan)}$$

$$\frac{}{\Gamma \vdash \mathrm{tt} \triangleright \mathsf{bool}} \quad \text{(T-Bool-true)} \qquad \frac{}{\Gamma \vdash \mathrm{ff} \triangleright \mathsf{bool}} \quad \text{(T-Bool-false)}$$

$$\frac{}{\Gamma \vdash 1 \triangleright \mathsf{nat}} \quad \text{(T-Nat)} \qquad \frac{\Gamma \vdash e \triangleright \mathsf{bool}}{\Gamma \vdash \mathtt{not}(e) \triangleright \mathsf{bool}} \quad \text{(T-Not)}$$

$$\frac{}{\Gamma \vdash \mathtt{arrived}\ k \triangleright \mathsf{bool}} \quad \text{(T-Session-Arrived)}$$

$$\frac{}{\Gamma \vdash \mathtt{arrived}\ a \triangleright \mathsf{bool}} \quad \text{(T-Shared-Arrived)}$$

$$\frac{\Sigma \text{ completed}}{\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma} \quad \text{(T-Nil)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma' \quad \Sigma' \leq \Sigma}{\Theta; \Gamma \vdash P \triangleright \Sigma} \quad \text{(T-Sub-s)}$$

$$\frac{\Gamma \vdash u \triangleright \langle S \rangle \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S}{\Theta; \Gamma \vdash u(x : S).P \triangleright \Sigma} \quad \text{(T-Accept)} \qquad \frac{\Gamma \vdash u \triangleright \langle S \rangle \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : \overline{S}}{\Theta; \Gamma \vdash \overline{u}(x : \overline{S}).P \triangleright \Sigma} \quad \text{(T-Request)}$$

$$\frac{\Gamma \vdash e \triangleright U \quad \Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S}{\Theta; \Gamma \vdash k\,!\langle e \rangle; P \triangleright \Sigma \cdot k : !\langle U \rangle; S} \quad \text{(T-Send-u)} \qquad \frac{\Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma \cdot k : S}{\Theta; \Gamma \vdash k\,?(x); P \triangleright \Sigma \cdot k : ?(U); S} \quad \text{(T-Receive-u)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S'}{\Theta; \Gamma \vdash k\,!\langle k' \rangle; P \triangleright \Sigma \cdot k : !\langle S \rangle; S' \cdot k' : S} \quad \text{(T-Send-s)} \qquad \frac{\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S' \cdot x : S}{\Theta; \Gamma \vdash k\,?(x); P \triangleright \Sigma \cdot k : ?(S); S'} \quad \text{(T-Receive-s)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S_i}{\Theta; \Gamma \vdash k \lhd l_i; P \triangleright \Sigma \cdot k : \oplus\{l_i : S_i\}_{i \in I}} \quad \text{(T-Select)}$$

$$\frac{\forall i.i \in I \quad \Theta; \Gamma \vdash P_i \triangleright \Sigma \cdot k : S_i}{\Theta; \Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : \&\{l_i : S_i\}_{i \in I}} \quad \text{(T-Branch)}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \quad \Theta; \Gamma \vdash P \triangleright \Sigma \quad \Theta; \Gamma \vdash Q \triangleright \Sigma}{\Theta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Sigma} \quad \text{(T-If)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Sigma \quad \Theta; \Gamma \vdash Q \triangleright \Sigma'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Sigma \cdot \Sigma'} \quad \text{(T-Par)}$$

$$\frac{\Theta; \Gamma \cdot u : \langle S \rangle \vdash P \triangleright \Sigma}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, u : \langle S \rangle)\ P \triangleright \Sigma} \quad \text{(T-Restr-a)}$$

$$\frac{\tilde{x} = \tilde{x}_U \cdot \tilde{x}_S \quad \Theta \cdot \mathbf{X} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot \tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S} \quad \Theta \cdot \mathbf{X} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash Q \triangleright \Sigma}{\Theta; \Gamma \vdash \mathtt{def}\ \mathbf{X}\langle \tilde{x} \rangle = P\ \mathtt{in}\ Q \triangleright \Sigma} \quad \text{(T-Def-in)}$$

$$\frac{\Sigma \text{ completed} \quad \tilde{x} = \tilde{x}_U \cdot \tilde{x}_S \quad \Gamma \vdash \tilde{x}_U \triangleright \tilde{U}}{\Theta \cdot \mathbf{X} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash \mathbf{X}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x}_S : \tilde{S}} \quad \text{(T-Process-var)}$$

$$\frac{\forall i.i \in I \quad \Theta; \Gamma \vdash P_i \triangleright \Sigma \cdot k : S_i' \quad \mathsf{match}(S_i', S_i) = \sigma_i}{\Theta; \Gamma \vdash \mathtt{typecase}\ k\ \mathtt{of}\ \{(\vec{\mathsf{X}})S_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : [S_i\sigma_i]_{i \in I}} \quad \text{(T-Typecase)}$$

Figure 4.9: Static Type System

$$(\Delta \odot \Delta')(s) = \begin{cases} (S, [S']\,\vec{\tau}) & \text{if } \Delta(s) = S, \Delta'(s) = [S']\,\vec{\tau}, \text{ and } S \leq S' \\ (S', [S]\,\vec{\tau}) & \text{if } \Delta(s) = [S]\,\vec{\tau}, \Delta'(s) = S', \text{ and } S' \leq S \\ \Delta(s) & \text{if } s \in \text{dom}(\Delta) \text{ and } s \notin \text{dom}(\Delta') \\ \Delta(s) & \text{if } s \in \text{dom}(\Delta') \text{ and } s \notin \text{dom}(\Delta) \\ undefined & \text{otherwise} \end{cases}$$

and is defined as

$$\Delta \odot \Delta' = \{s : (\Delta(s), \Delta'(s)) | s \in \text{dom}(\Delta) \cap \text{dom}(\Delta')\} \cup \Delta \backslash \text{dom}(\Delta') \cup \Delta' \backslash \text{dom}(\Delta)$$

When composing two environments we first take into account the shared endpoint configuration. We cannot have more than one shared endpoint for a shared channel. It is wrong to have more than one shared queues for a name because reduction can happen in any of the two queues resulting in errors in the computation. The composition for session requires that if a session mapping exists in one of the two sets then the session exists in the composition. If the session exists in the two composing sets, the one mapping is from the static typing system and the other is from the runtime typing system and the two corresponding sessions satisfy the subtyping relation then the mapping in the two sets consists of the session type along with the queue values.

4.6.3 Session Type Remainder

The Session Type Remainder S is calculated from a session type S' and a vector of queue types $\vec{\tau}$ ($S' - \vec{\tau} = S$) according to the following rules. It is an algebra on sessions and queue values (τ) that results in a session type. These operations are used to compare session types in runtime. This way we take into account the values in a queue and the session type of the queue that are evolving with computation.

$$\frac{}{S - \varepsilon = S} \quad \text{(Empty)}$$

$$\frac{S - \vec{\tau} = S'}{!\langle T\rangle; S - \vec{\tau} = !\langle T\rangle; S'} \quad \text{(Send)}$$

$$\frac{S - \vec{\tau} = S'}{?(T); S - T{\cdot}\vec{\tau} = S'} \quad \text{(Receive)}$$

$$\frac{1 \leq i \leq n, \quad S_i - \vec{\tau} = S'}{\oplus\{l_i : S_i\}_{i \in I} - \vec{\tau} = \oplus\{l_i : S'\}_{i \in I}} \quad \text{(Select)}$$

$$\frac{1 \leq i \leq n, \quad S_i - \vec{\tau} = S'}{\&\{l_i : S_i\}_{i \in I} - l_i{\cdot}\vec{\tau} = S'} \quad \text{(Branch)}$$

4.6.4 Runtime Typing Rules

Rule (Message-u) requires that values have type U, rule (Message-s) requires that sessions have type of session types while rule label requires that labels have the label type.

Rule (Ses-Config) types a session endpoint configuration. This rules types every value in the queue, the typing is composed by every individual typing and the mapping for sessions defined above. Rule (Restr-sess) checks for duality of dual endpoints. The session remainder algebra and subtyping are used for this purpose. Type duality is the heart of session types. When the two endpoints are dual the communication is consistent and occurs without problems. Rule (Par) composes two typings be the operation defined above. When we restrict a shared channel we make sure that the shared name exist in the typing and then we remove the shared name from the typing so it can be removed in another scope. This is expressed by rule (Restr-chan). Finally shared

$$\frac{\Gamma \vdash v \triangleright U \quad \Sigma \text{ completed}}{\Gamma; \Sigma \vdash v : U} \quad \text{(Message-u)}$$

$$\frac{}{\Gamma; s : S \vdash s : S} \quad \text{(Message-s)}$$

$$\frac{\Sigma \text{ completed}}{\Gamma; \Sigma \vdash l : l} \quad \text{(Label)}$$

$$\frac{\forall i. 1 \leq i \leq n \quad \Gamma; \Sigma_i \vdash h_i : \tau_i}{\Theta; \Gamma \vdash s\,[S] : h_1 \cdot \ldots \cdot h_n \triangleright (\Sigma_1 \cdot \ldots \cdot \Sigma_n) \odot s : [S]\,\tau_1 \cdot \ldots \cdot \tau_n} \quad \text{(Ses-Config)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau}_1) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau}_2) \quad S_i' = S_i - \vec{\tau}_i \quad i \in \{1, 2\} \quad S_1' \leq \overline{S_2'}}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, s)\ P \triangleright \Delta} \quad \text{(Restr-sess)}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \odot \Delta'} \quad \text{(Par)}$$

$$\frac{\Theta; \Gamma \cdot a : \langle S \rangle \vdash P \triangleright \Delta \cdot a}{\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a : \langle S \rangle)\ P \triangleright \Delta} \quad \text{(Restr-chan)}$$

$$\frac{\Gamma \vdash a \triangleright \langle S \rangle \quad \forall i. 1 \leq i \leq n \quad \Gamma; \Sigma_i \vdash \overline{s}_i : S_i \quad S_i \leq S}{\Theta; \Gamma \vdash a\,[\langle S \rangle] : s_1 \cdot \ldots \cdot s_n \triangleright (\Sigma_1 \cdot \ldots \cdot \Sigma_n \odot \overline{s}_1 : [S_1]\,\varepsilon \odot \cdots \odot \overline{s}_n : [S_n]\,\varepsilon) \odot a} \quad \text{(Sh-Config)}$$

Figure 4.10: Runtime Typing Rules

endpoint configuration typing is done with (Sh-Config). The types of each session in the queue are checked and are included in the construct typing along with the name of the shared channel. The missing endpoint typing is also composed in the typing.

Chapter 5

Event Driven Programs using the Session Type Event System

Examples are very important in order to understand the event model and the system that was defined previously. For this purpose we chose some event based examples to study and implement using the session type algebra developed in the previous chapter.

By giving examples we can draw conclusions about implementation techniques and patterns that arise in this kind of programming model. The system is being tested for the ability to describe application accurate and simple. Through examples we can also recommend changes in the system so it can make programming better.

It is very interesting to type these examples and draw conclusions about the type system and the session types that might occur from a program. We want to study possible patterns that appeared in different patterns and have a better view of the event model we have formalized.

5.1 Programming Examples

In this section we can see programs defining application that are using the event driven programming model. The main example demonstrated here is a web application and especially web servers. We begin with simple trivial examples and we proceed with more complex examples in order to see how the formalism can describe this king of applications.

Before we begin I want to give the guidelines for reading the diagrams in this chapter. Rectangular shapes with a name in it give a named process. Green interrupted lines show session request towards the arrow. Blue lines show data sending towards the arrow. Red interrupted lines show process reduction towards the arrow. Session endpoints are shown as a queue structure. Note that not all endpoints that can be created are shown in the diagrams.

5.1.1 Simple Webserver

A web server is probably the best example that can be model using events. A web server is responsible for handling different requests from different clients. Requests and interaction with clients can be seen as events occurring during the web server computation. The next example is a trivial example and it is used as reference for the definitions that follow. It receives requests from clients and responds with a file.

The program below describes a web server using the event driven model. The **Acceptor** process accepts connections in the form of session channels. The **Handler** process handles the interaction in an event driven fashion. Using the arrived and typecase construct it checks the status and then the type of an endpoint and acts according to the state of the session - event. Endpoints are shown in the form of a queue notation.

$$
\begin{array}{l}
\texttt{def} \\
\quad \mathbf{Acceptor}\langle x_a xy\rangle = \texttt{arrived}\ x_a\ \texttt{then}\ x_a(s).x\,!\langle s\rangle; \mathbf{Handler}\langle x_a xy\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Handler}\langle x_a xy\rangle \\
\\
\quad \mathbf{Handler}\langle x_a xy\rangle = \texttt{arrived}\ y\ \texttt{then} \\
\qquad\qquad y\,?(z); \texttt{arrived}\ z\ \texttt{then}\ \texttt{typecase}\ z\ \texttt{of}\ \{ \\
\qquad\qquad\qquad ?(Req); !\langle Doc\rangle; \texttt{end} : \\
\qquad\qquad\qquad\quad z\,?(req); z\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a xy\rangle \\
\qquad\qquad\qquad \} \\
\qquad\qquad\qquad \texttt{else}\ x\,!\langle z\rangle; \mathbf{Acceptor}\langle x_a xy\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Acceptor}\langle x_a xy\rangle \\
\\
\texttt{in} \\
\quad \overline{b}(x).b(y).\mathbf{Acceptor}\langle axy\rangle \mid a\,[\langle S\rangle] : \varepsilon \mid b\,[\langle S'\rangle] : \varepsilon
\end{array}
$$

We can see two parts of computation. The **Acceptor** process accepts a connection, if there is a connection, and stores the new session in an event queue and proceeds to **Handler** process. If there is not an entry in the shared endpoint, it proceeds to the **Handler** process where different events can be processed. **Handler** process gets a session from the event queue, if the queue is not empty, typechecks the event and process the client request. After processing, it proceeds to the **Acceptor** process. If the queue is empty then it also evolves in the **Acceptor** process.

5.1.2 Cache Server

The next example is a more complex web server. The scenario is that a client requests from a server a document. The server is using a **Cache** process to retrieve documents and send them to the clients. The web server initiates sessions between two processes, the client and the **Cache** process. Handling the two sessions as events requires a more complex typecase definition. Notice that whenever the handler is waiting for an event to proceed, it stores the event context and retrieves the next event for processing. Here we can see the whole event model philosophy in action.

Figure 5.1 shows a diagram of the server. The structure of the web server is simple. Processes **Acceptor**, **Handler** are the two processes that define the server. The **Client** process requests for a new session. **Acceptor** process accepts the session and stores it the session endpoint. The **Handler** process receives an event/session from the private session endpoint and data from the **Client** and responds to the **Client** with the requested file.

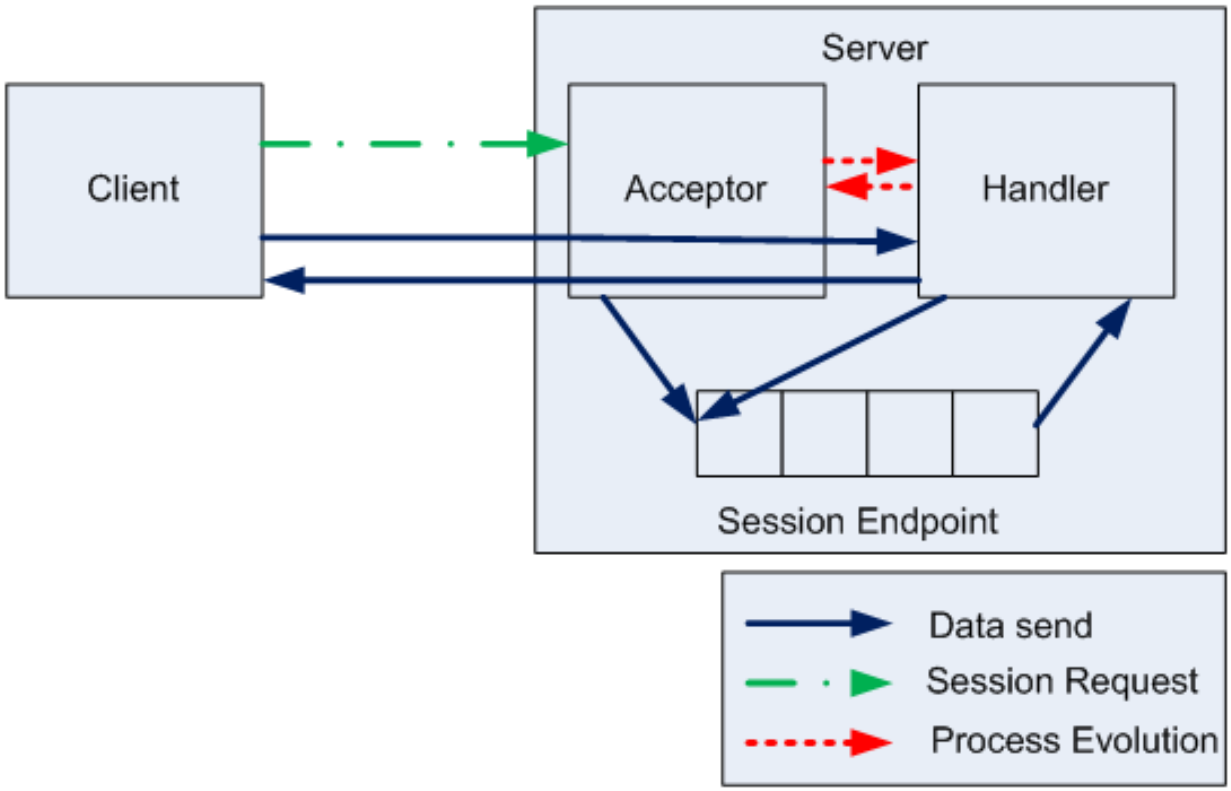

Figure 5.1: Simple Server

$$
\begin{array}{l}
\texttt{def} \\
\quad \mathbf{Acceptor}\langle x_a x_c x y\rangle = \texttt{arrived}\ x_a\ \texttt{then}\ x_a(s).x\,!\langle s\ null\rangle; \mathbf{Handler}\langle x_a x_c x y\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Handler}\langle x_a x_c x y\rangle \\
\\
\quad \mathbf{Handler}\langle x_a x_c x y\rangle = \texttt{arrived}\ y\ \texttt{then} \\
\qquad\qquad y\,?(z z'); \texttt{arrived}\ z\ \texttt{then}\ \texttt{typecase}\ z\ \texttt{of}\ \{ \\
\qquad\qquad\quad ?(Req); !\langle Doc\rangle; \texttt{end}: \\
\qquad\qquad\qquad \texttt{typecase}\ z'\ \texttt{of}\ \{ \\
\qquad\qquad\qquad \texttt{end}: \\
\qquad\qquad\qquad z\,?(req); \overline{x_c}(s).s\,!\langle req\rangle; \\
\qquad\qquad\qquad \texttt{arrived}\ s\ \texttt{then}\ s\,?(doc); z\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c x y\rangle \\
\qquad\qquad\qquad\ \texttt{else}\ x\,!\langle sz\rangle; \mathbf{Acceptor}\langle x_a x y\rangle \\
\qquad\qquad\qquad \} \\
\qquad\qquad\quad ?(Doc); \texttt{end}\,!\langle Doc\rangle; \texttt{end}: \\
\qquad\qquad\qquad \texttt{typecase}\ z'\ \texttt{of}\ \{ \\
\qquad\qquad\qquad !\langle Doc\rangle; \texttt{end}: \\
\qquad\qquad\qquad z\,?(doc); z'\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c x y\rangle \\
\qquad\qquad\qquad \} \\
\qquad\qquad\quad \} \\
\qquad\qquad\quad \texttt{else}\ x\,!\langle z z'\rangle; \mathbf{Acceptor}\langle x_a x_c x y\rangle \\
\qquad\quad \texttt{else}\ \mathbf{Acceptor}\langle x_a x_c x y\rangle
\end{array}
$$

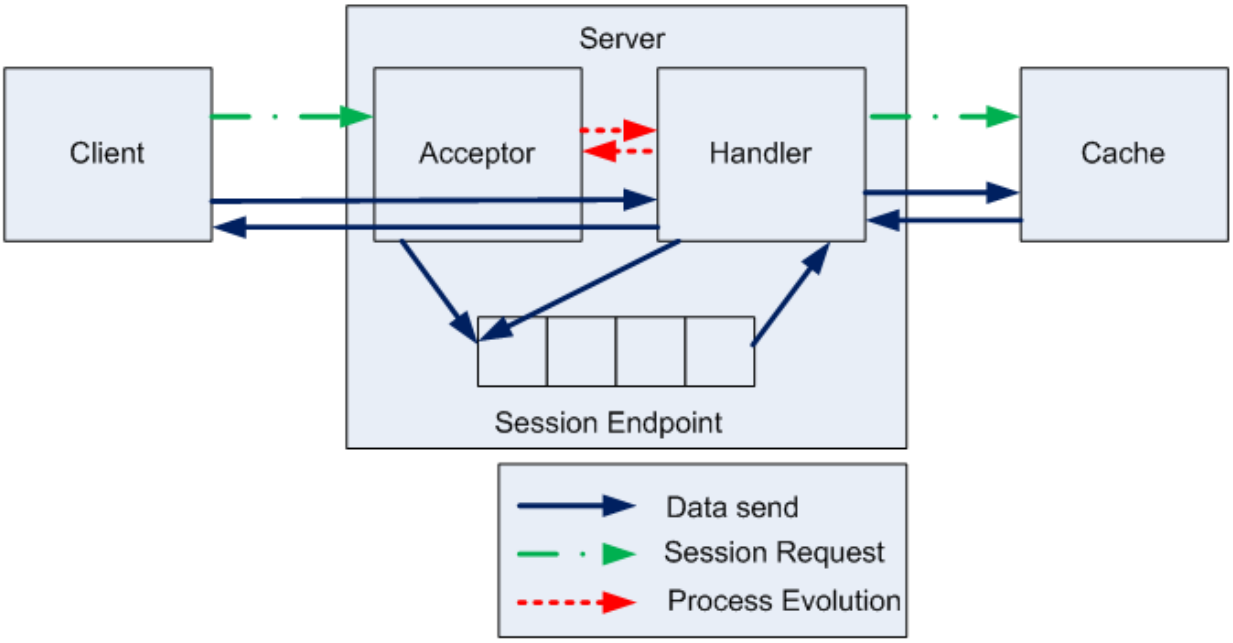

Figure 5.2: Cache Server

$$\begin{aligned}\textbf{Cache}\langle x_c\rangle = x_c(x).(&\textbf{Cache}\langle x_c\rangle \mid x\,?(req);\texttt{if } req == 1 \texttt{ then } x\,!\langle doc_1\rangle;\mathbf{0}\\ &\texttt{else if } doc == 2 \texttt{ then } \ldots\\ &\vdots\\ &\texttt{else } x\,!\langle error\rangle;\mathbf{0}\end{aligned}$$

$$\texttt{in}\quad \overline{b}(x).b(y).\textbf{Acceptor}\langle acxy\rangle \mid \textbf{Cache}\langle c\rangle \mid a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid \ \mid c\,[\langle S_c\rangle] : \varepsilon$$

The **Acceptor** process accepts requests and proceeds to the **Handler** process. The **Handler** process gets the next event, finds its type and processes it. If the event is a client request, it initiates a session with the **Cache** process and sends the request. If the **Cache** process has not responded yet, it stores the context in a session endpoint and continues with computation. If the event is waiting for a response from the **Cache** process then it receives the response and sends the document to the client. In all cases after finishing computation it proceeds to the **Acceptor** process to check for new requests from clients.

The **Cache** process simply accepts a new session and spawns a new parallel process that reads the request and replies with a document.

If we compare this implementation with the simple server implementation we can see similarities in the structure of the application. We have the same **Acceptor** process that does the same computation each time, to accept connection from clients, regardless the session type of the connection. Again we have a **Handler** process that handles events with a typecase construct. The checking for message arriving and the interaction with the session endpoint configuration also presents similarities.

Figure 5.2 shows a diagram of the cache server. We see the **Client** requesting to the **Acceptor**. The **Acceptor** stores sessions in an session endpoint and it evolves to **Handler** process, which initiates a session and exchanges data with the **Cache** process. It also receives and sends data to the session endpoint and it can evolve back to the **Acceptor** process. An event finishes when the **Handler** process sends a respond to the **Client**. **Cache** process accepts requests from a **Handler** process, receives data and sends data to the **Handler** process.

5.1.3 SEDA Cache Webserver

This example uses the Staged Event Driven Architecture approach to implement a web server. This approach is a pure event driven approach that breaks the application to components and pipelines the components based on events that can happen. Here events that can happen are the session passing between processes.

This approach accepts a request from a client. Then after recognizing the event it forwards the whole event context to a cache structure. The cache finds the requested document and forwards it to a send process. The send process simply replies to the client with the requested document.

$$
\begin{array}{l}
\texttt{def} \\
\quad \mathbf{Acceptor}\langle x_a xyy'\rangle = \texttt{arrived}\ x_a\ \texttt{then}\ x_a(s).x\,!\langle s\rangle; \mathbf{Handler}\langle x_a xyy'\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Handler}\langle x_a xyy'\rangle \\
\\
\quad \mathbf{Handler}\langle x_a xyy'\rangle = \texttt{arrived}\ y\ \texttt{then} \\
\qquad\qquad y\,?(z); \texttt{arrived}\ z\ \texttt{then}\ \texttt{typecase}\ z\ \texttt{of}\ \{ \\
\qquad\qquad\qquad ?(Get); !\langle Doc\rangle; \texttt{end} : \\
\qquad\qquad\qquad\quad z\,?(req); y'\,!\langle z\ req\rangle; \mathbf{Acceptor}\langle x_a xyy'\rangle \\
\qquad\qquad\qquad \} \\
\qquad\qquad\qquad \texttt{else}\ x\,!\langle z\rangle; \mathbf{Acceptor}\langle x_a xyy'\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Acceptor}\langle x_a xyy'\rangle \\
\\
\quad \mathbf{Cache}\langle xy\rangle = x\,?(z\ req); \\
\qquad\qquad\qquad \texttt{if}\ req == 1\ \texttt{then}\ y\,!\langle z\ doc_1\rangle; \mathbf{Cache}\langle xy\rangle \\
\qquad\qquad\qquad\ \texttt{else if}\ req == 2\ \texttt{then}\ y\,!\langle z\ doc_2\rangle; \mathbf{Cache}\langle xy\rangle \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\ \texttt{else}\ y\,!\langle z\ error\rangle; \mathbf{Cache}\langle yz\rangle \\
\\
\quad \mathbf{Send}\langle x\rangle = x\,?(z\ doc); z\,!\langle doc\rangle; \mathbf{Send}\langle x\rangle \\
\texttt{in} \\
\quad \overline{b}(s).b(y).\overline{c}(s').\mathbf{Acceptor}\langle asys'\rangle \mid c(x).\overline{d}(s'').\mathbf{Cache}\langle xs''\rangle \mid d(y).\mathbf{Send}\langle y\rangle \\
\quad a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \mid d\,[\langle S_d\rangle] : \varepsilon
\end{array}
$$

Again we have defined the **Acceptor** and **Handler** processes with a similar structure as the previous examples. The difference here is that the **Handler** process after typecase, delegates the session and passes the session context, in this case the request, to **Cache** process. Storing the event in the session endpoint only happens when waiting for the request to arrive. The **Cache** process simply reads a new session and finds the requested document. Again the session and session context, in this case the requested document, are passed to the **Send** process. The send process after receiving from the **Cache** process it simply replies the document to the client via the delegated session.

Figure 5.3 shows the seda cache server. We can see the pipeline of the application as a network. The data flows through session endpoints between the different components. The acceptor handler architecture is still the same as the previous examples.

5.1.4 SEDA Mail Server

The most complicated so far example is a Mail Server. This server can receive requests for sending an email and requests for reading emails. From this implementation some details are missing such as storing and retrieving mail. Following the SEDA we use sessions/events to interact between processes and here we can see the use of the typecase construct handling more than one event.

We have chosen an example like this one to demonstrate a more complex pipeline example. In the previous example we have a straightforward pipeline. Here the pipeline is split in two directions, according to the event happening.

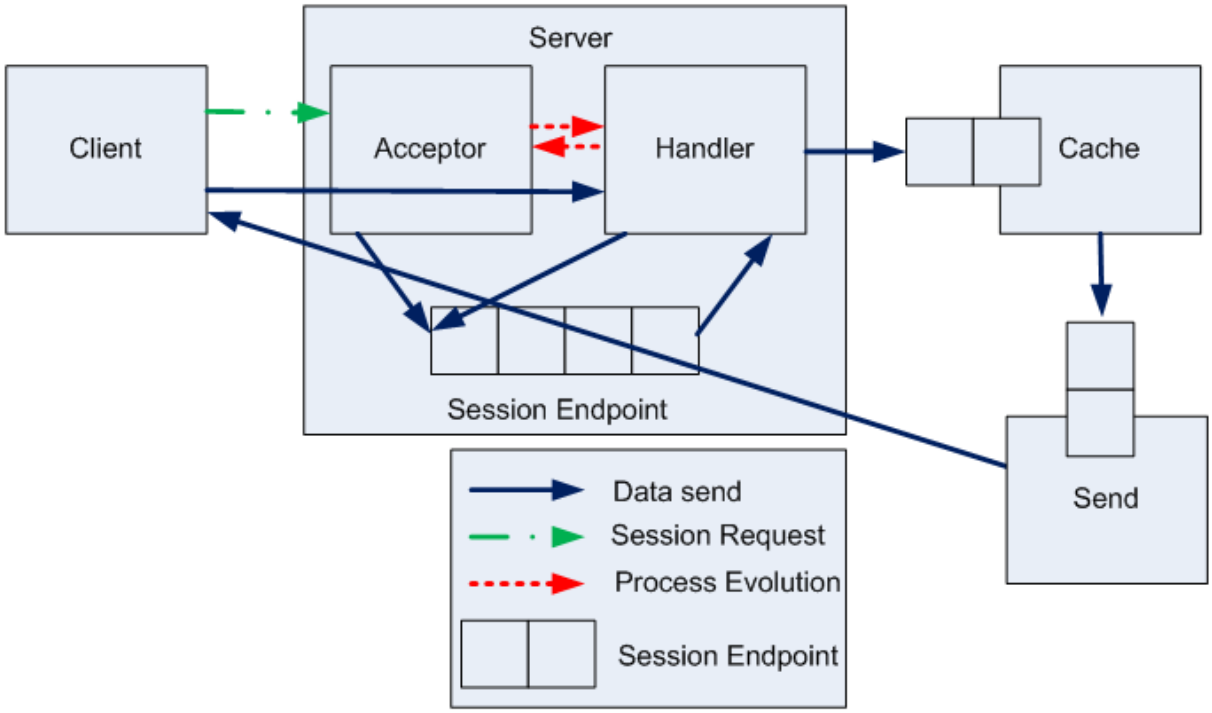

Figure 5.3: SEDA Cache Server

$$
\begin{aligned}
&\texttt{def}\\
&\quad \mathbf{Acceptor}\langle x_a xyy'\rangle = \texttt{arrived}\ x_a\ \texttt{then}\ x_a(s).x\,!\langle s\rangle; \mathbf{Handler}\langle x_a xyy_c y_a\rangle\\
&\qquad\qquad \texttt{else}\ \mathbf{Handler}\langle x_a xyy_c y_a\rangle\\
&\\
&\quad \mathbf{Handler}\langle x_a xyy_c y_a\rangle = \texttt{arrived}\ y\ \texttt{then}\\
&\qquad\qquad y\,?(z); \texttt{arrived}\ z\ \texttt{then typecase}\ z\ \texttt{of}\ \{\\
&\qquad\qquad\qquad ?(Log); !\langle Mail\rangle; \texttt{end}:\\
&\qquad\qquad\qquad\quad z\,?(log); y_c\,!\langle z\ log\rangle; \mathbf{Acceptor}\langle x_a xyy_c y_a\rangle\\
&\qquad\qquad\qquad ?(Addr); ?(Email); \texttt{end}:\\
&\qquad\qquad\qquad\quad z\,?(addr); y_a\,!\langle z\ addr\rangle; \mathbf{Acceptor}\langle x_a xyy_c y_a\rangle\\
&\qquad\qquad\qquad \}\\
&\qquad\qquad\qquad \texttt{else}\ x\,!\langle z\rangle; \mathbf{Acceptor}\langle x_a xyy_c y_a\rangle\\
&\qquad\qquad \texttt{else}\ \mathbf{Acceptor}\langle x_a xyy_c y_a\rangle\\
&\\
&\quad \mathbf{Cache}\langle xy\rangle = y\,?(z\ log);\\
&\qquad\qquad \texttt{if}\ log == 1\ \texttt{then}\ x\,!\langle z\ mail_1\rangle; \mathbf{Cache}\langle xy\rangle\\
&\qquad\qquad\ \texttt{else if}\ log == 2\ \texttt{then}\ x\,!\langle z\ mail_2\rangle; \mathbf{Cache}\langle xy\rangle\\
&\qquad\qquad \vdots\\
&\qquad\qquad\ \texttt{else}\ x\,!\langle z\ error\rangle; \mathbf{Cache}\langle yz\rangle
\end{aligned}
$$

$$\mathbf{Send}\langle x\rangle = x\,?(z\ doc); z\,!\langle doc\rangle; \mathbf{Send}\langle x\rangle$$

$$\mathbf{AccAddr}\langle xyy_a\rangle = \texttt{arrived}\ y_a\ \texttt{then}\ y_a\,?(z\ addr); x\,!\langle z\ addr\rangle; \mathbf{Addr}\langle xyy_a\rangle$$
$$\texttt{else}\ \mathbf{Addr}\langle xyy_a\rangle$$

$$\mathbf{Addr}\langle xyy_a\rangle = \texttt{arrived}\ y\ \texttt{then}$$
$$y\,?(z\ addr); \texttt{arrived}\ z\ \texttt{then typecase}\ z\ \texttt{of}\ \{$$
$$?(Email); \texttt{end}:$$
$$z\,?(email); \mathbf{AccAddr}\langle xyy_a\rangle$$
$$\}$$
$$\texttt{else}\ x\,!\langle z\rangle; \mathbf{AccAddr}\langle xyy_a\rangle$$
$$\texttt{else}\ \mathbf{AccAddr}\langle xyy_a\rangle$$

in

$$\overline{b}(s).b(y).\overline{c}(s_c).\overline{d}(s_a).\overline{c}(s').\mathbf{Acceptor}\langle asys_cs_a\rangle \mid c(x).\overline{e}(s_s).\mathbf{Cache}\langle xs_s\rangle \mid$$
$$e(y).\mathbf{Send}\langle y\rangle \mid \overline{f}(s'').f(y).d(y_a).\mathbf{Addr}\langle s''yy_a\rangle \mid$$
$$a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \mid d\,[\langle S_d\rangle] : \varepsilon \mid e\,[\langle S_e\rangle] : \varepsilon \mid f\,[\langle S_f\rangle] : \varepsilon$$

We can see the same definition of the **Acceptor** process. The difference here is that this process can accept different session types on its shared channel, one session type for sending mail and another for retrieving. To interleave between **Acceptor** and **Handler** processes, the program follows identical approach as the previous examples. The new element here is that the pipeline breaks after the **Handler** process. The one direction of the pipeline goes through **Cache** and **Send** process, following the same pattern as the previous example, only this time a login is required instead of a request and instead of a document the client's mail is sent. The second direction is more interesting. The **Handler** process delegates the session context in **Addr** process. This process is waiting to receive email. Interaction between waiting and receiving is done by periodically evolving to the **AccAddr** process, just like the pair **Acceptor**, **Handler** processes.

5.1.5 Travel Agency

A different program than the web server approach is presented here. This is a travel agency example and it was initially introduced to describe session types in [14]. Here we tailor travel agency to the event driven programming model. Specifically in order to have multiple events we allow the travel agency to serve as many clients as possible. This is an important example because it introduces all the notions of this algebra: event driven handling, recursion and recursive session types and branching.

The scenario description wants customers to initiate sessions with a travel agency. Through a loop the customers are informed about the prices of the destinations they request. Finally if an agreement is reach the agency delegates the session to a service that schedules the customer's holidays on the desired date.

$$
\begin{array}{l}
\texttt{def} \\
\quad \mathbf{Acceptor}\langle x_a x_s x y\rangle = \texttt{arrived}\ x_a\ \texttt{then}\ x_a(s).x\,!\langle s\ null\rangle; \mathbf{Handler}\langle x_a x_s x y\rangle \\
\qquad\qquad \texttt{else}\ \mathbf{Handler}\langle x_a x_s x y\rangle \\
\\
\quad \mathbf{Handler}\langle x_s x y\rangle = \texttt{arrived}\ y\ \texttt{then} \\
\qquad\qquad y\,?(z\ dest); \texttt{arrived}\ z\ \texttt{then typecase}\ z\ \texttt{of}\ \{ \\
\qquad\qquad\qquad \mu\mathbf{t}.?(Dest); !\langle Price\rangle; \&\{accept : ?(Addr\ Date); !\langle Date\rangle; \texttt{end}, repeat : \mathbf{t}, reject : \texttt{end}\} : \\
\qquad\qquad\qquad\quad z\,?(dest); z\,!\langle price\rangle; \\
\qquad\qquad\qquad\quad \texttt{arrived}\ z\ \texttt{then}\ z \triangleright \{ \\
\qquad\qquad\qquad\qquad accept : \overline{x_s}(s).s\,!\langle z\ dest\rangle; \mathbf{Acceptor}\langle x_a x_s x y\rangle, \\
\qquad\qquad\qquad\qquad repeat : y\,?(z\ dest); \mathbf{Acceptor}\langle x_a x_s x y\rangle, \\
\qquad\qquad\qquad\qquad reject : \mathbf{Acceptor}\langle x_a x_s x y\rangle \\
\qquad\qquad\qquad\quad \} \\
\qquad\qquad\qquad\quad \texttt{else}\ y\,?(z\ dest); \mathbf{Acceptor}\langle x_a x_s x y\rangle \\
\qquad\qquad\qquad \mu\mathbf{t}.\&\{accept : ?(Addr\ Date); !\langle Date\rangle; \texttt{end}, repeat : ?(Dest); !\langle Price\rangle;\ \mathbf{t}, reject : \texttt{end}\} : \\
\qquad\qquad\qquad\quad z \triangleright \{ \\
\qquad\qquad\qquad\qquad accept : \overline{x_s}(s).s\,!\langle z\ dest\rangle; \mathbf{Acceptor}\langle x_a x_s x y\rangle, \\
\qquad\qquad\qquad\qquad repeat : y\,?(z\ dest); \mathbf{Acceptor}\langle x_a x_s x y\rangle, \\
\qquad\qquad\qquad\qquad reject : \mathbf{Acceptor}\langle x_a x_s x y\rangle \\
\qquad\qquad\qquad\quad \} \\
\qquad\qquad \} \\
\qquad\qquad \texttt{else}\ y\,?(z\ dest); \mathbf{Acceptor}\langle x_a x_s x y\rangle \\
\\
\quad \mathbf{Service}\langle x_s\rangle = x_s(x).(\mathbf{Service}\langle x_s\rangle \mid x\,?(addr\ date); x\,!\langle date'\rangle; \mathbf{0} \\
\\
\texttt{in} \\
\quad \overline{b}(x).b(y).\mathbf{Acceptor}\langle acxy\rangle \mid \mathbf{Service}\langle c\rangle \mid a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid\ \mid c\,[\langle S_c\rangle] : \varepsilon
\end{array}
$$

We again have the same server architecture when it comes to accept and handle requests. The difference here is the recursive interaction between customer and agency. The customer makes a branch choice. The choice can either terminate the session, initiates a delegation procedure or recurse. The recursion here is dealed by storing the session in the private endpoint and by proceeding to see if there are any new requests. This is a design choice we made for this program. We could just as easily let the **Handler** process to continue to interact with the customer. Notice the types in the typecase construct. The recursion definition requires for the types to be consistent in both cases. This is fixed if we notice the typing definition of the repeat label.

The travel agency figure is shown in 5.4. We can see that the architecture of this diagram has similarities with the cache server diagram. Here we have the recursive interaction between Handler and customer and also interaction between customer and service. We can see session initiation request going from customer to travel agency and from travel agency to service. Again even though we have a different program than the previous program the acceptor handler architecture is the same.

5.2 Typing of Programs

Next we proceed with typing some of the examples in this chapter. Typing is equally interesting with the examples because it reveals the typing patterns for similar implementations and design choices.

To avoid repetition and save space the typing of the same process will be omitted whenever is possible. If the reader finds a derivation tree incomplete then the details missing should be part of another derivation tree or already been typed as a part of the same derivation tree.

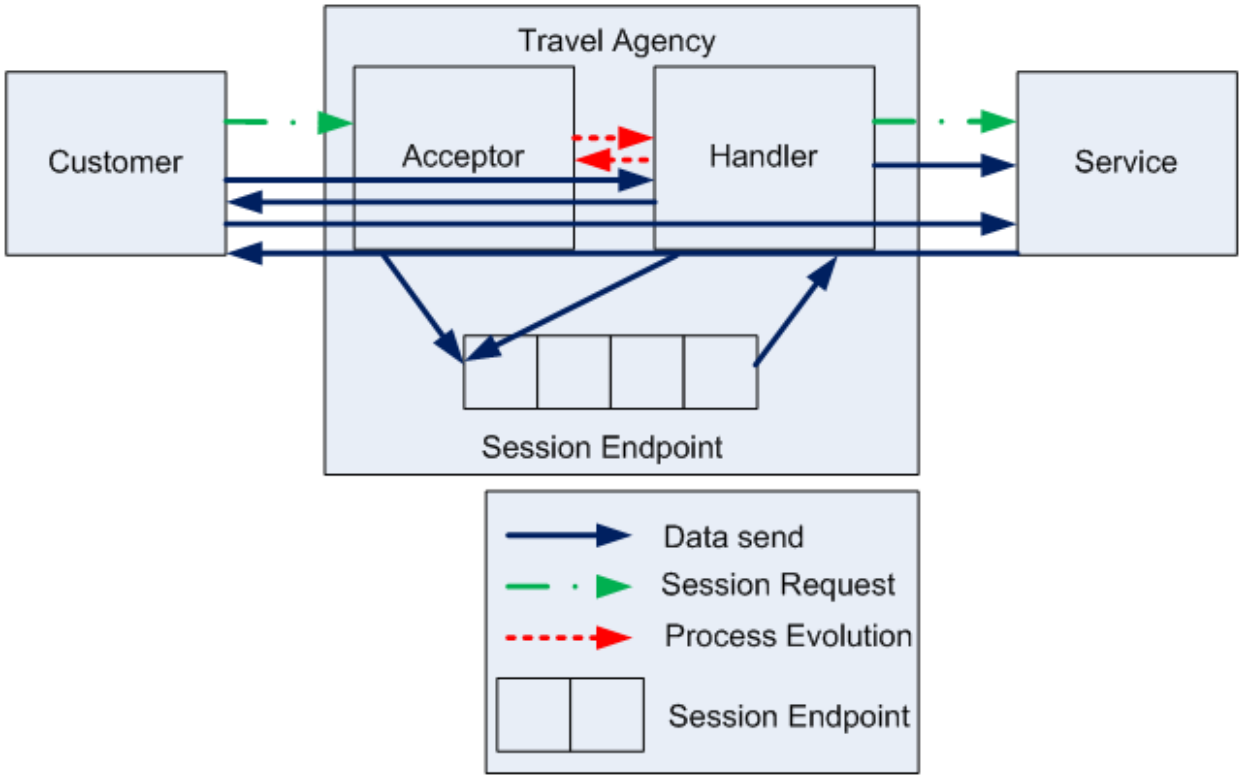

Figure 5.4: Travel Agency Server

5.2.1 Simple Web Server Typing

Following the typing for the Simple Web Server (5.1.1). Again this typing will serve as a reference point for other typings.

We begin with the typing of the **Acceptor** process.

$$\dfrac{\dfrac{\dfrac{\Sigma \;\; \text{completed} \quad x_a xy : \langle S'' \rangle S S' \quad \Gamma \vdash x_a \triangleright \langle S'' \rangle}{\Theta; \Gamma \vdash \mathbf{Handler}\langle x_a xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S'} \quad \Gamma \vdash x_a \triangleright \langle S'' \rangle}{\Theta; \Gamma \vdash x_a(z).x\,!\langle z \rangle; \mathbf{Handler}\langle x_a xy \rangle \triangleright \Sigma \cdot x :!\langle S'' \rangle; S \cdot y : S' \cdot z : S''}}{\Theta; \Gamma \vdash \texttt{arrived}\; x_a \;\texttt{then}\; \ldots \;\texttt{else}\; \ldots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle S'' \rangle; \mathbf{t} \cdot y : S'}$$

This typing has revealed a very interesting observation about endpoints used to store event context. This typing shows a recursive session type. The recursive session type comes up because the process has the form of $\texttt{if}\; e \;\texttt{then}\; P \;\texttt{else}\; Q$. The typing rule ((T-If)) wants P and Q to have the same type. If we apply this to processes $x_a(z).x\,!\langle z \rangle; \mathbf{Handler}\langle x_a xy \rangle$ and $\mathbf{Handler}\langle x_a xy \rangle$ have:

$$\begin{aligned} x &:!\langle S'' \rangle; S \\ x &: S \end{aligned}$$

The equation here that comes from (T-if) rule is:

$$!\langle S'' \rangle; S = S \Rightarrow S = \mu \mathbf{t}.!\langle S'' \rangle; \mathbf{t}$$

The solution of the equation gives the recursive typing for the session endpoint configuration. The recursive session type is not surprising at all. We expect that kind of typing for a queue used to continuously and arbitrary storing and retrieving (In the **Acceptor** case only storing happens) session types and session contexts. The interesting part is that the recursion is revealed through the

equation that comes from the (T-If) rule. We expect this to happen if we continue typing examples.

The Handler process is typed next:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad x_a xy : \langle S'' \rangle S S' \quad \Gamma \vdash x_a \triangleright \langle S'' \rangle}{\Gamma \vdash doc \triangleright Doc \quad \Theta; \Gamma' \vdash \mathbf{Acceptor}\langle x_a xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S'}}{\Theta; \Gamma' \cdot reg : Get \vdash z\,!\langle doc \rangle; \mathbf{Acceptor}\langle x_a xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S' \cdot z :!\langle Doc \rangle; \mathtt{end}}}{\Theta; \Gamma \vdash z\,?(req); z\,!\langle doc \rangle; \mathbf{Acceptor}\langle x_a xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S' \cdot z :?(Get); !\langle Doc \rangle; \mathtt{end}}}{\Theta; \Gamma \vdash \mathtt{typecase}\ z\ \mathtt{of}\ \{\dots\} \triangleright \Sigma \cdot x : S \cdot y : S' \cdot z : [?(Get); !\langle Doc \rangle; \mathtt{end}]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad uxy : \langle S'' \rangle S S' \quad \Gamma \vdash u \triangleright \langle S'' \rangle}{\Theta; \Gamma \vdash \mathbf{Acceptor}\langle x_a xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S'}}{\Theta; \Gamma \vdash x\,!\langle z \rangle; \mathbf{Acceptor}\langle x_a xy \rangle \triangleright \Sigma \cdot x :!\langle S'' \rangle; S \cdot y : S' \cdot z : S_z}}{\Theta; \Gamma \vdash \mathtt{arrived}\ z\ \mathtt{then}\ \dots\ \mathtt{else}\ \dots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \cdot y : S' \cdot z : [?(Get); !\langle Doc \rangle; \mathtt{end}]}}{\Theta; \Gamma \vdash y\,?(z); \mathtt{arrived}\ z\ \mathtt{then}\ \dots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \cdot y :?([?(Get); !\langle Doc \rangle; \mathtt{end}]); S'}}{\Theta; \Gamma \vdash \mathtt{arrived}\ y\ \mathtt{then}\ y\,?(z); \dots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \cdot y : \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t}}$$

Again, as expected, the typing for the private session endpoints has recursive types. The type for storing is a recursive sending type and the type for retrieving is recursive receiving type. The details and the equations for the recursive typing are the same as the typing of the **Acceptor** process.

Agent Instance typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad axy : \langle [?(Get);\ !\langle Doc \rangle; \mathtt{end}] \rangle \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \quad \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t} \quad \Gamma \vdash u \triangleright \langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle}{\Theta; \Gamma \vdash \mathbf{Acceptor}\langle axy \rangle \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \cdot y : \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t} \quad \Gamma \vdash b \triangleright \langle \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t} \rangle}}{\Theta; \Gamma \vdash b(y).\mathbf{Acceptor}\langle axy \rangle \triangleright \Delta \cdot x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t} \quad \Gamma \vdash b \triangleright \langle \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t} \rangle}}{\Theta; \Gamma \vdash \overline{b}(x).b(y).b(y).\mathbf{Acceptor}\langle axy \rangle \triangleright \Sigma}$$

$$\dfrac{\dfrac{\Gamma \vdash a \triangleright \langle S \rangle \quad \Gamma \vdash b \triangleright \langle S' \rangle \quad \Theta; \Gamma; \Sigma \vdash \varepsilon : \varepsilon}{\Theta; \Gamma \vdash a\,[S] : \varepsilon \mid b\,[S'] : \varepsilon \triangleright a \odot b}}{\Theta; \Gamma \vdash \overline{b}(x).b(y).\mathbf{Acceptor}\langle axy \rangle \mid a\,[S] : \varepsilon \mid b\,[S'] : \varepsilon \triangleright \Delta}$$

The typings of the channels in the instance of process **Acceptor** come from the typing of the definition of processes **Acceptor** and **Handler** in the body of the recursive definition. In more detail from process **Handler** we have that:

$$x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t}$$
$$y : \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t}$$

Process $\mathbf{Handler}\langle x_a xy \rangle$ is evolved to process $\mathbf{Acceptor}\langle x_a xy \rangle$ carrying the $x_a xy$ variables along the evolvement, so the above typing holds for process **Acceptor** as well. As a direct result we have that:

$$\Gamma \vdash x_a \triangleright \langle [?(Get);\ !\langle Doc \rangle; \mathtt{end}] \rangle$$

The instance **Acceptor**$\langle axy \rangle$, following the typing in the body of the recursive definition has session typing:

$$x : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc \rangle; \mathtt{end}] \rangle; \mathbf{t}$$
$$y : \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t}$$

and shared name judgements:

$$\Gamma \vdash a \triangleright \langle [?(Get);\ !\langle Doc \rangle; \mathtt{end}] \rangle$$
$$\Gamma \vdash b \triangleright \langle \mu \mathbf{t}.?([?(Get); !\langle Doc \rangle; \mathtt{end}]); \mathbf{t} \rangle$$

In the final derivation we notice that the application has a runtime typing judgement. This follows from the fact that we used the runtime typing rule (Sh-Config) to type the instance of the shared endpoint configuration. Finally in the parallel composition we use the runtime rule (Par) because we deal with shared endpoint configuration structures. Summing up we have that:

$$\Delta = \Sigma \odot a \odot b$$

5.2.2 Cache Server

The Cache Server example (section 5.1.2) has more complicated typing than the first server. The most interesting fact here is typing the typecase construct. In this example the typecase construct has more than one case. Also we would like to see the recursive typing of the private session endpoints.

Acceptor process typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \ \text{ completed} \quad x_a x_c xy : \langle S_a \rangle \langle S_c \rangle S S' \quad \Gamma \vdash x_a x_c \triangleright \langle S_a \rangle \langle S_c \rangle}{\Theta; \Gamma \vdash \mathbf{Handler} \langle x_a x_c xy \rangle \triangleright \Sigma \cdot x : S \cdot y : S'} \quad \Gamma \vdash x_a \triangleright \langle S_a \rangle \quad \Theta; \Gamma \vdash x\,!\langle z \rangle; \mathbf{Handler} \langle x_a x_c xy \rangle \triangleright \Sigma \cdot x :!\langle S'' \rangle; S \cdot y : S' \cdot z : S''}{\Theta; \Gamma \vdash x_a(z).x\,!\langle z \rangle; \mathbf{Handler} \langle x_a x_c xy \rangle \triangleright \Sigma \cdot x :!\langle S'' \rangle; S \cdot y : S'}}{\Theta; \Gamma \vdash \mathtt{arrived}\ x_a\ \mathtt{then}\ \ldots\ \mathtt{else}\ \ldots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle S'' \rangle; \mathbf{t} \cdot y : S'}$$

This typing has the same typing as the process **Acceptor** for the simple server. Minor changes consider the variables in the process's definition.

Process **Handler** typing:

$$\frac{\Sigma \text{ completed} \quad x_a x_c xy : \langle S_a\rangle\langle S_c\rangle S S' \quad \Gamma \vdash x_a x_c \triangleright \langle S_a\rangle\langle S_c\rangle}{\Gamma \vdash doc \triangleright Doc \quad \Theta; \Gamma' \vdash \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'}$$
$$\frac{}{\Theta; \Gamma'\cdot doc : Doc \vdash z\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'\cdot z :!\langle Doc\rangle; \mathtt{end}}$$
$$\Theta; \Gamma \vdash s\,?(doc); z\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'\cdot z :!\langle Doc\rangle; \mathtt{end}\cdot s :?(Doc); \mathtt{end}$$

$$\frac{\Theta; \Gamma \vdash x\,!\langle sz\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x :!\langle S_s S_z\rangle; S\cdot y : S'\cdot s : S_s\cdot z : S_z}{\Gamma \vdash reg \triangleright Get \quad \Theta; \Gamma \vdash \mathtt{arrived}\ s\ \mathtt{then}\ \ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot s :?(Doc); \mathtt{end}\cdot z :!\langle Doc\rangle; \mathtt{end}}$$
$$\frac{}{\Theta; \Gamma \vdash s\,!\langle req\rangle; \mathtt{arrived}\ s\ \mathtt{then}\ \ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot s :!\langle Get\rangle; ?(Doc); \mathtt{end}\cdot z :!\langle Doc\rangle; \mathtt{end} \quad \Gamma \vdash x_c \triangleright \langle ?(Get); !\langle Doc\rangle; \mathtt{end}\rangle}$$
$$\frac{}{\Theta; \Gamma'\cdot reg : Get \vdash \overline{c}(s).s\,!\langle req\rangle; \ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :!\langle Doc\rangle; \mathtt{end}}$$
$$\frac{}{\Theta; \Gamma \vdash z\,?(req); \overline{c}(s).\ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Get); !\langle Doc\rangle; \mathtt{end}}$$
$$\frac{}{\Theta; \Gamma \vdash \mathtt{typecase}\ z'\ \mathtt{of}\ \{\mathtt{end} : \ldots\} \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Get); !\langle Doc\rangle; \mathtt{end}\cdot z' : [\mathtt{end}] \quad [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}] \leq [\mathtt{end}]}$$
$$\Theta; \Gamma \vdash \mathtt{typecase}\ z'\ \mathtt{of}\ \{\mathtt{end} : \ldots\} \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Get); !\langle Doc\rangle; \mathtt{end}\cdot z' : [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]$$

$$\frac{\Gamma \vdash doc \triangleright Doc \quad \Theta; \Gamma' \vdash \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'}{\Theta; \Gamma'\cdot doc : Doc \vdash z'\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'\cdot z' :!\langle Doc\rangle; \mathtt{end}}$$
$$\frac{}{\Theta; \Gamma \vdash z\,?(doc); z'\,!\langle doc\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x : S\cdot y : S'\cdot z' :!\langle Doc\rangle; \mathtt{end}\cdot z :?(Doc); \mathtt{end}}$$
$$\frac{}{\Theta; \Gamma \vdash \mathtt{typecase}\ z'\ \mathtt{of}\ \{z\,?(doc); \ldots\} \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Doc); \mathtt{end}\cdot z' : [!\langle Doc\rangle; \mathtt{end}] \quad [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}] \leq [\mathtt{end}]}$$
$$\frac{}{\Theta; \Gamma \vdash \mathtt{typecase}\ z'\ \mathtt{of}\ \{z\,?(doc); \ldots\} \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Doc); \mathtt{end}\cdot z' : [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]}$$
$$\frac{}{\Theta; \Gamma \vdash \mathtt{typecase}\ z\ \mathtt{of}\ \{\ldots\} \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle ?(Doc); \mathtt{end}\ !\langle Doc\rangle; \mathtt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z : [?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\cdot z' : [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]}$$

$$\frac{\Theta; \Gamma \vdash x\,!\langle zz'\rangle; \mathbf{Acceptor}\langle x_a x_c xy\rangle \triangleright \Sigma\cdot x :!\langle S_z S_{z'}\rangle; S\cdot y : S'\cdot z : S_z\cdot z' : S_{z'}}{\Theta; \Gamma \vdash \mathtt{arrived}\ y\ \mathtt{then}\ \mathtt{typecase}\ z\ \mathtt{of}\ \{\ldots\}\ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\ [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]\rangle; \mathbf{t}\cdot y : S'\cdot z : [?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\cdot z' : [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]}$$
$$\frac{}{\Theta; \Gamma \vdash y\,?(zz'); \mathtt{arrived}\ y\ \mathtt{then}\ \ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\ [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]\rangle; \mathbf{t}\cdot y :?([?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\ [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]); S'}$$
$$\Theta; \Gamma \vdash \mathtt{arrived}\ y\ \mathtt{then}\ y\,?(zz'); \ldots\ \mathtt{else}\ \ldots \triangleright \Sigma\cdot x : \mu\mathbf{t}.!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\ [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]\rangle; \mathbf{t}\cdot y : \mu\mathbf{t}.?([?(Get); !\langle Doc\rangle; \mathtt{end}, ?(Doc); \mathtt{end}]\ [\mathtt{end}, !\langle Doc\rangle; \mathtt{end}]); \mathbf{t}$$

The interesting fact about the above typing is the typing of the complex typecase process $\mathtt{typecase}\ z\ \mathtt{of}\ \{\ldots\}$. In order to type this process we need to see the typings of processes $\mathtt{typecase}\ z'\ \mathtt{of}\ \{\mathtt{end} : z\,?(req); \ldots\}$ and $\mathtt{typecase}\ z'\ \mathtt{of}\ \{!\langle Doc\rangle; \mathtt{end} : \ldots\}$:

$$x : \mu\mathbf{t}.!\langle ?(Doc); \texttt{end}\ !\langle Doc\rangle; \texttt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Get); !\langle Doc\rangle; \texttt{end}\cdot z' : [\texttt{end}]$$
$$x : \mu\mathbf{t}.!\langle ?(Doc); \texttt{end}\ !\langle Doc\rangle; \texttt{end}\rangle; \mathbf{t}\cdot y : S'\cdot z :?(Doc); \texttt{end}\cdot z' : [!\langle Doc\rangle; \texttt{end}]$$

Typing for session z' is different in the two processes. Rule for typecase wants the two typings of z' to be the same since z' is not the session being typecased here. To overcome this problem we use the subtyping relation to find a direct supertype of the session z' type in the two typecase processes. Actually the same thing happens with the typecased session z. A direct supertype of the two types is the session set type that comes up when we combine the two session types, exactly like the typing of the typecased channel. This case is described by rule (T-Sub). (T-Sub) applies to both processes before rule (T-Typecase) is applied to the whole typecase process.

Process **Cache** typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \text{ completed}}{\Gamma \vdash doc_i \triangleright Doc \quad \Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma}}{\Theta; \Gamma\cdot reg : Get \vdash x\,!\langle doc_i\rangle; \mathbf{0} \triangleright \Sigma\cdot x :!\langle Doc\rangle; \texttt{end}}}{\Theta; \Gamma \vdash x\,?(req); \texttt{if}\ req == i\ \texttt{then}\ x\,!\langle doc_i\rangle; \mathbf{0} \triangleright \Sigma\cdot x :?(Get); !\langle Doc\rangle; \texttt{end}}}{\Gamma \vdash x_c \triangleright \langle ?(Get); !\langle Doc\rangle; \texttt{end}\rangle}}{\begin{array}{c}\Theta; \Gamma \vdash \mathbf{Cache}\langle x_c\rangle \mid x\,?(req); \texttt{if}\ req == i\ \texttt{then}\ x\,!\langle doc_i\rangle; \mathbf{0} \triangleright \Sigma \\ x :?(Get); !\langle Doc\rangle; \texttt{end}\end{array}}}{\Theta; \Gamma \vdash x_c(x).(\mathbf{Cache}\langle x_c\rangle \mid x\,?(req); \texttt{if}\ req == i\ \texttt{then}\ x\,!\langle doc_i\rangle; \mathbf{0}) \triangleright \Sigma}$$

This typing is rather straightforward so some details like the typing of process Variable **Cache** and the parallel composition are omitted.

Agent Instance typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \text{ completed}}{\begin{array}{c}acxy : \langle S_a\rangle\langle S_c\rangle\mu\mathbf{t}.!\langle[?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]\rangle; \mathbf{t} \\ \mu\mathbf{t}.?([?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]); \mathbf{t} \\ \Gamma \vdash ac \triangleright \langle S_a\rangle\langle S_c\rangle\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash \mathbf{Acceptor}\langle acxy\rangle \triangleright \Sigma \\ x : \mu\mathbf{t}.!\langle[?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]\rangle; \mathbf{t} \\ y : \mu\mathbf{t}.?([?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]); \mathbf{t} \\ \Gamma \vdash b \triangleright \langle \mu\mathbf{t}.?([?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]); \mathbf{t}\rangle\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash b(y).\mathbf{Acceptor}\langle acxy\rangle \triangleright \Sigma \\ x : \mu\mathbf{t}.!\langle[?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]\rangle; \mathbf{t} \\ \Gamma \vdash b \triangleright \langle \mu\mathbf{t}.?([?(Get); !\langle Doc\rangle; \texttt{end}, ?(Doc); \texttt{end}]\ [\texttt{end}, !\langle Doc\rangle; \texttt{end}]); \mathbf{t}\rangle\end{array}}}{\Theta; \Gamma \vdash \overline{b}(x).b(y).\mathbf{Acceptor}\langle acxy\rangle \triangleright \Sigma}$$

$$\dfrac{\begin{array}{c}\Sigma \text{ completed} \quad c : \langle S_c\rangle \\ \Gamma \vdash c \triangleright \langle ?(Get); !\langle Doc\rangle; \texttt{end}\rangle\end{array}}{\Theta; \Gamma \vdash \mathbf{Cache}\langle c\rangle \triangleright \Sigma}$$

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash abc \triangleright \langle S_a\rangle\langle S_b\rangle\langle S_c\rangle \\ \Theta; \Gamma; \Sigma_{\{a,b,c\}} \vdash \varepsilon : \varepsilon\end{array}}{\Theta; \Gamma \vdash a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \triangleright a\cdot b\cdot c}}{\begin{array}{c}\Theta; \Gamma \vdash \overline{b}(x).b(y).\mathbf{Acceptor}\langle acxy\rangle \mid \mathbf{Cache}\langle c\rangle \mid a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \triangleright \\ \Sigma \odot a \odot b \odot c\end{array}}$$

5.2.3 SEDA Cache Server

SEDA applications rely on event pipelining for computation. The typing of such pipelines presents great interest, especially for the session types that are used for process communication inside the application. The session typing for private session endpoints and the typecase construct are expected to be typed as the examples typed above.

The SEDA cache server (5.1.3) has the same specification and interface as the cache server (5.1.2), but different implementation architecture. The typing follows:

Acceptor process typing:

$$\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad x_a xyy' : \langle S_a \rangle \langle S_c \rangle S_x S_y S_{y'} \quad \Gamma \vdash x_a \triangleright \langle S_a \rangle}{\Theta; \Gamma \vdash \mathbf{Handler}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x : S_x \cdot y : S_y \cdot y : S_{y'}}}{\begin{array}{c}\Gamma \vdash x_a \triangleright \langle S_a \rangle \\ \Theta; \Gamma \vdash x\,!\langle z \rangle; \mathbf{Handler}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x :!\langle S_z \rangle; S_x \cdot y : S_y \cdot y : S_{y'} \cdot z : S_z\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash x_a(z).x\,!\langle z \rangle; \mathbf{Handler}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x :!\langle S_z \rangle; S_x \cdot y : S' \cdot y : S_{y'}\end{array}}$$

$$\Theta; \Gamma \vdash \texttt{arrived}\ x_a\ \texttt{then}\ x_a(z).x\,!\langle z \rangle; \mathbf{Handler}\langle x_a xyy' \rangle\ \texttt{else}\ \mathbf{Handler}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle S_z \rangle; \mathbf{t} \cdot y : S_y \cdot y : S_{y'}$$

This typing is a small extension to the **Acceptor** typing in the cache server typing. The extension follows the presence of the session y' typing in the overall process typing.

Handler process typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad x_a xyy' : \langle S_a \rangle \langle S_c \rangle S_x S_y S_{y'} \quad \Gamma \vdash x_a \triangleright \langle S_a \rangle}{\Gamma \vdash req \triangleright Get \quad \Theta; \Gamma' \vdash \mathbf{Acceptor}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x : S_x \cdot y : S_y \cdot y' : S_{y'}}}{\begin{array}{c}\Theta; \Gamma' \cdot reg : Get \vdash y'\,!\langle z\ req \rangle; \mathbf{Acceptor}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x : S_x \cdot \\ y : S_y \cdot z : S_z \cdot y' :!\langle S_z\ Get \rangle; S_{y'}\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash z\,?(req); y'\,!\langle z\ req \rangle; \mathbf{Acceptor}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x : S_x \cdot \\ y : S_y \cdot z :?(Get); S_z \cdot y' :!\langle S_z\ Get \rangle; S_{y'}\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash \texttt{typecase}\ z\ \texttt{of}\ \{?(Get); !\langle Doc \rangle; \texttt{end} : \ldots\} \triangleright \Sigma \cdot x : S_x \cdot \\ y : S_y \cdot z : [?(Get); S_z] \cdot y' :!\langle S_z\ Get \rangle; S_{y'}\end{array}}$$

$$\dfrac{\dfrac{\Theta; \Gamma \vdash x\,!\langle z \rangle; \mathbf{Acceptor}\langle x_a xyy' \rangle \triangleright \Sigma \cdot x :!\langle S'_z \rangle; S_x \cdot y : S_y \cdot y' : S_{y'} \cdot z : S_z}{\begin{array}{c}\Theta; \Gamma \vdash \texttt{arrived}\ z\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); S_z] \rangle; \mathbf{t} \cdot \\ y : S_y \cdot y' : \mu \mathbf{t}.!\langle S_z\ Get \rangle; \mathbf{t} \cdot z : [?(Get); S_z]\end{array}}}{\begin{array}{c}\Theta; \Gamma \vdash y\,?(z); \texttt{arrived}\ z\ \texttt{then}\ \ldots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); S_z] \rangle; \mathbf{t} \cdot \\ y :?([?(Get); S_z]); S_y \cdot y' : \mu \mathbf{t}.!\langle S_z\ Get \rangle; \mathbf{t}\end{array}}$$

$$\Theta; \Gamma \vdash \texttt{arrived}\ y\ \texttt{then}\ y\,?(z); \ldots \triangleright \Sigma \cdot x : \mu \mathbf{t}.!\langle [?(Get); S_z] \rangle; \mathbf{t} \cdot y : \mu \mathbf{t}.?([?(Get); S_z]); \mathbf{t} \cdot y' : \mu \mathbf{t}.!\langle S_z\ Get \rangle; \mathbf{t}$$

This is a straightforward typing where we observe many of the conclusions described in previous typing examples.

Cache process Typing:

$$\dfrac{\dfrac{\dfrac{\Sigma \;\text{ completed} \quad xy : S_x S_y}{\Gamma \vdash doc_i \triangleright Doc \quad \Theta; \Gamma \vdash \mathbf{Cache}\langle xy \rangle \triangleright \Sigma \cdot x : S_x \cdot y : S_y}}{\Theta; \Gamma \cdot reg : Get \vdash y\,!\langle z\ doc_i \rangle; \mathbf{Cache}\langle xy \rangle \triangleright \Sigma \cdot y :!\langle S_z\ Doc \rangle; S_y \cdot x : S_x \cdot z : S_z}}{\Theta; \Gamma \vdash x\,?(z\ req); y\,!\langle z\ doc_i \rangle; \mathbf{Cache}\langle xy \rangle \triangleright \Sigma \cdot y :!\langle S_z\ Doc \rangle; S_y \cdot x :?(S_z\ Get); S_x}$$

From the last judgement we notice the typing of the process variable session channels. Since the definition of **Cache** is recursive, meaning that **Cache** refers to itself in its definition, we can state and solve the following equations:

$$S_y = !\langle S_z\ Doc\rangle; S_y \Rightarrow S_y = \mu \mathbf{t}.!\langle S_z\ Doc\rangle; \mathbf{t}$$
$$S_x = ?(S_z\ Get); S_x \Rightarrow S_x = \mu \mathbf{t}.?(S_z\ Get); \mathbf{t}$$

We can see the recursion appearing once again in sessions between architectural components. There is no way determining the recursion by the body of the definition, but we can see recursion if we observe the whole process definition.
Send process typing:

$$\frac{\frac{\frac{\Delta \ \text{completed} \quad x : S}{\Gamma \vdash doc \triangleright Doc \quad \Theta; \Gamma \vdash \mathbf{Send}\langle x\rangle \triangleright \Sigma \cdot x : S}}{\Theta; \Gamma \cdot doc : Doc \vdash z\,!\langle doc\rangle; \mathbf{Send}\langle x\rangle \triangleright \Sigma \cdot x : S \cdot z :!\langle Doc\rangle; \mathtt{end}}}{\Theta; \Gamma' \vdash x\,?(z\ doc); z\,!\langle doc\rangle; \mathbf{Send}\langle x\rangle \triangleright \Sigma \cdot x :?(!\langle Doc\rangle; \mathtt{end}\ Doc); S}$$

The same argument as the **Cache** process reveals recursion so:

$x : \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Doc); \mathbf{t}$

Recursion Instance typing:

$$\frac{\frac{\frac{\frac{\Sigma \ \text{completed} \quad asys' : \langle a\rangle S_s S_y S_{s'}}{\Gamma \vdash a \triangleright \langle ?(Get); !\langle Doc\rangle; \mathtt{end}\rangle}}{\Theta; \Gamma \vdash \mathbf{Acceptor}\langle asys'\rangle \triangleright \Sigma \cdot s : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}]\rangle; \mathbf{t} \cdot y : \mu \mathbf{t}.?([?(Get); !\langle Doc\rangle; \mathtt{end}]); \mathbf{t} \cdot s' : \mu \mathbf{t}.!\langle !\langle Doc\rangle; \mathtt{end}\ Get\rangle; \mathbf{t}}}{\Gamma \vdash c \triangleright \langle \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Get); \mathbf{t}\rangle}}{\Theta; \Gamma \vdash \overline{c}(s').\mathbf{Acceptor}\langle asys'\rangle \triangleright \Sigma \cdot s : \mu \mathbf{t}.!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}]\rangle; \mathbf{t} \cdot y : \mu \mathbf{t}.?([?(Get); !\langle Doc\rangle; \mathtt{end}]); \mathbf{t}}}{\Gamma \vdash b \triangleright \langle \mu \mathbf{t}.?([?(Get); !\langle Doc\rangle; \mathtt{end}]); \mathbf{t}\rangle}}{\Theta; \Gamma \vdash b(y).\overline{c}(s').\mathbf{Acceptor}\langle asys'\rangle \triangleright \Sigma s :!\langle [?(Get); !\langle Doc\rangle; \mathtt{end}]\rangle; s' :!\langle !\langle Doc\rangle; \mathtt{end}\ Get\rangle; \Gamma \vdash b \triangleright \langle \mu \mathbf{t}.?([?(Get); !\langle Doc\rangle; \mathtt{end}]); \mathbf{t}\rangle}$$

$$\frac{\frac{\frac{\Sigma \ \text{completed} \quad xs'' : S_x S_{s''}}{\Theta; \Gamma \vdash \mathbf{Cache}\langle xs''\rangle \triangleright \Sigma \cdot x : \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Get); \mathbf{t} \cdot s'' : \mu \mathbf{t}.!\langle !\langle Doc\rangle; \mathtt{end}\ Doc\rangle; \mathbf{t}}}{\Gamma \vdash d \triangleright \langle \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Doc); \mathbf{t}\rangle}}{\Theta; \Gamma \vdash \overline{d}(s'').\mathbf{Cache}\langle xs''\rangle \triangleright \Sigma \cdot x : \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Get); \mathbf{t} \quad \Gamma \vdash c \triangleright \langle \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Get); \mathbf{t}\rangle}}{\Theta; \Gamma \vdash c(x).\overline{d}(s'').\mathbf{Cache}\langle xs''\rangle \triangleright \Sigma}$$

$$\frac{\frac{\Sigma \ \text{completed} \quad y : S_y}{\Theta; \Gamma \vdash \mathbf{Send}\langle y\rangle \triangleright \Sigma \cdot y : \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Doc); \mathbf{t}}}{\Gamma \vdash d \triangleright \langle \mu \mathbf{t}.?(!\langle Doc\rangle; \mathtt{end}\ Doc); \mathbf{t}\rangle}}{\Theta; \Gamma \vdash d(y).\mathbf{Send}\langle y\rangle \triangleright \Sigma}$$

$$\frac{\frac{\Gamma \vdash abcd \triangleright \langle S_a\rangle\langle S_b\rangle\langle S_c\rangle\langle S_d\rangle \quad \Theta;\Gamma;\Sigma_{\{a,b,c,d\}} \vdash \varepsilon : \varepsilon}{\Theta;\Gamma \vdash a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \mid d\,[\langle S_d\rangle] : \varepsilon \triangleright a{\cdot}b{\cdot}c{\cdot}d}}{\begin{array}{c}\Theta;\Gamma \vdash \overline{b}(c).b(y).\overline{c}(s').\mathbf{Acceptor}\langle asys'\rangle \mid c(x).\overline{d}(s'').\mathbf{Cache}\langle xs''\rangle \mid d(y).\mathbf{Send}\langle y\rangle \mid \\ a\,[\langle S_a\rangle] : \varepsilon \mid b\,[\langle S_b\rangle] : \varepsilon \mid c\,[\langle S_c\rangle] : \varepsilon \mid d\,[\langle S_d\rangle] : \varepsilon \triangleright \Sigma \odot a \odot b \odot c \odot d\end{array}}$$

5.2.4 SEDA Mail Server

Following the patterns on the previous examples we can type the SEDA mail server. Here we only show the types of the typing environment of the program.

$a : \langle[?(Log); !\langle Mail\rangle; \mathtt{end}, ?(Addr); ?(Email);]\rangle$
$b : \langle \mu\mathbf{t}.?([?(Log); !\langle Mail\rangle; \mathtt{end}, ?(Addr); ?(Email);]); \mathbf{t}\rangle$
$c : \langle \mu\mathbf{t}.?(!\langle Mail\ Log\rangle; \mathtt{end}); \mathbf{t}\rangle$
$d : \langle \mu\mathbf{t}.?(?(Email\ Addr); \mathtt{end}); \mathbf{t}\rangle$
$e : \langle \mu\mathbf{t}.?(!\langle Mail\rangle;\ Mail); \mathtt{end}\rangle\mathbf{t}$
$f : \langle \mu\mathbf{t}.?(?(Email\ Addr); \mathtt{end}); \mathbf{t}\rangle$

5.2.5 Travel Agency Server

We give the typing of the environment.

$a : \langle \mu\mathbf{t}.?(Dest); !\langle Price\rangle; \&\{accept :?(Addr\ Date); !\langle Date\rangle; \mathtt{end}, repeat : \mathbf{t}, reject : \mathtt{end}\}\rangle$
$b : \langle \mu\mathbf{t}\text{'}.?([\mu\mathbf{t}.?(Dest); !\langle Price\rangle; \&\{accept :?(Addr\ Date); !\langle Date\rangle; \mathtt{end}, repeat : \mathbf{t}, reject : \mathtt{end}\}, \mu\mathbf{t}.\&\{accept :$
$?(Addr\ Date); !\langle Date\rangle; \mathtt{end}, repeat : ?(Dest); !\langle Price\rangle;\ \mathbf{t}, reject : \mathtt{end}\}]); \mathbf{t}\text{'}\rangle$
$c : \langle ?(?(Addr\ Date); !\langle Date\rangle; \mathtt{end}); \mathtt{end}\rangle$

The most interesting observation here is the nested recursion. We have recursion due to the select - branch structure and recursion due to the private endpoint. In the recursive endpoint we store a recursive typing. This is true when we have events that require to loop in order to get processed. The way we store and retrieving events also requires recursion.

5.3 Typecase Construct and Subtyping

In this section we consider the typing of the nested typecase example in the Cache Server (5.1.2). As explained in the typing we use subtyping in order to be able to type this process. In the typecase process session channel z' has a different in the two process cases of the typecase construct. To overcome the problem we use set subtyping and rule (T-Sub). This example seems to reveal possible use of subtyping in cases such as (T-Typecase), (T-If), (T-Branch) etc where we need more than one processes to have the same typing. To become more formal, consider:
$\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma{\cdot}k : S_i{\cdot}k' : S'_i$

If we combine the above processes in a typecase construct we have:
$\mathtt{typecase}\ k\ \mathtt{of}\ \{S_i : P_i\}_{i\in I}$

To be able to type the above process we need to use subtyping (rule (T-Sub)):

$\forall i \in I, S'_i \leq [S'_i]_{i\in I}$ by using rule (T-Sub) and the initial given processes we have $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma{\cdot}k : S_i{\cdot}k' : [S'_i]_{i\in I}$

Now we can type the typecase process:

$\Theta; \Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{S_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : [S_i]_{i \in I} \cdot k' : [S'_i]_{i \in I}$

The technique is similar if we need to deal with rules (T-Branch) and (T-If).

This observation helps us to understand an important fact about typecase typing ((T-Typecase)). This rule indirectly uses set subtyping in order for the program to have consistent session typing. The most important contribution of typecase is not the set subtyping but the matching that is done at runtime between the event/session and the cases of the typecase, in order to decide the process continuation. Set subtyping is just the solution given to the problem of typing multiple events in one channel.

As we so in the server example and in this discussion, multiple sessions can be found not just in sessions that are matched in a typecase construct but also in other situation. The typing of these sessions can be done by using the (T-Sub) rule and the set session type and the subtyping relation.

Chapter 6

Properties of the Event Driven Session Type System

As with all formal system, our system should satisfy certain properties. These properties complete our system as a computational model and state that we safely use it to specify applications.

Over the years theoreticians described complete systems as sound systems. Soundness is demonstrated through a Subject Reduction theorem. Subject Reduction wants two important properties to hold for a reduction system:

1. *Progress*: Any well typed specification cannot reduce to an error.

2. *Preservation* Any well typed specification can be reduced and the result is also well typed.

The first property says that a well typed program can always progress and terminates, while the second says that a well type program that can reduce it produces again a well type program.

In this chapter we will state a Subject Reduction theorem that contains the above intuitions and we will prove it in order to show the soundness of our system. As we will see in order to have a well type specification is not enough to have the typing of the specification but a few more properties need to hold.

We begin our discussion by giving key definitions. After that we claim and proof lemmas that are used in Subject Reduction proof. The Subject Reduction Theorem and its proof are next. The chapter finishes with a discussion on error states and the conditions that must hold so we can avoid them.

6.1 Basic Definitions

Before we begin the exploration of the properties holding in the session type system we must give a set of definitions relevant to those properties.

Definition 6.1.1 *Well Configured Session*

A runtime session typing Δ is *well configured* with respect to a session s if the following holds:

$\Delta = \Delta' \odot s : (S_1, [S_1']\,\vec{\tau_1}) \odot \overline{s} : (S_2, [S_2']\,\vec{\tau_2})$ implies

$S_i' \leq S_i, S_i'' = S_i - \vec{\tau_i}, i \in \{1,2\}, S_1'' \leq \overline{S_2''}$

We write *wc(Δ,s)* read as Δ is *well configured* with respect to s.

■

Definition 6.1.2 *Well configured Typing*

Δ is *well configured* if $\forall s \in dom(\Delta), wc(\Delta, s)$

■

Definition 6.1.3 *Ordering*

A relation over the set of runtime typings it is called an *ordering,*$\Delta \sqsubseteq \Delta'$, if the following holds:

1. $s :!\langle T\rangle; S \odot \overline{s} : [S']\,\vec{\tau} \sqsubseteq s : S \odot \overline{s} : [S']\,\vec{\tau}\cdot T$
2. $s :?(T); S \odot s : [?(T); S]\,T\cdot\vec{\tau} \sqsubseteq s : S \odot s : [S]\,\vec{\tau}$
3. $s : \&\{l_i : S_i\}_{i\in I} \odot s : [S]\,\vec{\tau} \sqsubseteq s : S_i \odot s : [S]\,\vec{\tau}\cdot l_i$
4. $s : \oplus\{l_i : S_i\}_{i\in I} \odot s : [\oplus\{l_i : S_i\}_{i\in I}]\,l_i\cdot\vec{\tau} \sqsubseteq s : S_i \odot s : [S_i]\,\vec{\tau}$
5. $s : \mu \boldsymbol{t}.S \odot s' : [S']\,\vec{\tau} \sqsubseteq s : S\{\mu \boldsymbol{t}.S/t\} \odot s' : [S'']\,\vec{\tau'}$ if $s : S \odot s' : [S']\,\vec{\tau} \sqsubseteq s : S\{\mu \boldsymbol{t}.S/t\} \odot s' : [S'']\,\vec{\tau'} s' = s$ or $s' = \overline{s}$
6. $\emptyset \sqsubseteq \emptyset$
7. $\Delta \odot \Delta_1 \sqsubseteq \Delta \odot \Delta_2$ if $\Delta_1 \sqsubseteq \Delta_2$ and $\Delta \odot \Delta_1$ defined

■

6.2 Key Lemmas

Before we proceed with the proof for Subject Reduction theorem we first need to prove some lemmas that are needed in the proof for Subject Reduction. All the lemmas in these section state properties of the typing system and the construction of the algebra and are used for stating and proving the Subject Reduction property.

The notations used in the proves are:

$\overset{I.H.}{\Longrightarrow}$: Apply the Induction hypothesis to the data on the left hand to get the conclusions on the right hand
$\overset{rulename}{\Longrightarrow}$: Apply the rule above the arrow to the data on the left hand to get the conclusions on the right hand
$\dot{\imath}_{\in I}\Sigma_i = \Sigma_1\cdot\ldots\cdot\Sigma_n, I = \{1,\ldots,n\}$
$\odot_{i\in I}\Delta_i = \Delta_1\cdot\ldots\cdot\Delta_n, I = \{1,\ldots,n\}$

6.2.1 Weakening Lemma

The weakening lemma states that if we have a process typing then we can *weaken* a typing mapping, in a sense that we can arbitraly add information under conditions, in the mapping and preserve the typing of the process. Formally:

Lemma 6.2.1 *Weakening*

Assume $\Theta; \Gamma \vdash P \triangleright \Sigma$

1. If $\boldsymbol{X} \notin dom(\Theta)$ then $\Theta\cdot\boldsymbol{X} : T; \Gamma \vdash P \triangleright \Sigma$
2. If $u \notin dom(\Gamma)$ then $\Theta; \Gamma\cdot u : U \vdash P \triangleright \Sigma$
3. If $k \notin dom(\Sigma)$ and $S = \boldsymbol{end}$ or $S = \bot$then $\Theta; \Gamma \vdash P \triangleright \Sigma\cdot k : S$

The proof requires for induction on the structure of the typing tree. The typing tree is formed by the static typing system. By using the last rule applied for typechecking P we can prove the above parts of the lemma. Note that the leaves of the typing tree for process are always formed by rules (T-Nill) and (T-Process-var). These rules will form the basic step in the induction. The inductive hypothesis comes from the lemma preposition. The inductive step is straightforward on the typing rules that construct processes.

Proof.

Part 1

Basic Step:

Case (T-Nill) Trivial. Let $\mathbf{X} \notin \mathrm{dom}(\Theta)$. From case rule we assume that Σ is completed, so $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$ and $\Theta \cdot \mathbf{X} : T; \Gamma \vdash \mathbf{0} \triangleright \Sigma$

Case (T-Process-var) Trivial. Let $\mathbf{X} \notin \mathrm{dom}(\Theta)$ and $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash \mathbf{Y}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{x}_S$. From case rule we assume that Σ completed, $\tilde{x} = \tilde{x}_U \cdot \tilde{x}_S, \Gamma \vdash \tilde{x}_U \triangleright \tilde{U}$. By case rule we conclude that $\Theta \mathbf{Y} : \tilde{U} \cdot \tilde{S} \cdot \mathbf{X} : T; \Gamma \vdash \mathbf{Y}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{x}_S$

Induction Step:

For all cases assume that $\mathbf{X} \notin \mathrm{dom}(\Theta)$

Case (T-Sub-s) Let $\Theta; \Gamma \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma$

Case (T-Accept) Let $\Theta; \Gamma \vdash u(x : S).P \triangleright \Sigma$ then by (T-Accept) we have that $\Gamma \vdash u \triangleright \langle S \rangle$ and $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot x : S \overset{T-Accept}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash u(x : S).P \triangleright \Sigma$

Case (T-Request) Let $\Theta; \Gamma \vdash \overline{u}(x : S).P \triangleright \Sigma$ then by (T-Request) we have that $\Gamma \vdash u \triangleright \langle \overline{S} \rangle$ and $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot x : S \overset{T-Request}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash \overline{u}(x : S).P \triangleright \Sigma$

Case (T-Send-u) Let $\Theta; \Gamma \vdash k\,!\langle e \rangle; P \triangleright \Sigma \cdot k :!\langle U \rangle; S$ then by (T-Send-u) we have that $\Gamma \vdash e \triangleright U$ and $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot k : S \overset{T-Send-u}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k\,!\langle e \rangle; P \triangleright \Sigma \cdot k :!\langle U \rangle; S$

Case (T-Receive-u) Let $\Theta; \Gamma \vdash k\,?(x); P \triangleright \Sigma \cdot k :?(U); S$ then by (T-Send-u) we have that $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma \cdot k : S \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \cdot x : U \vdash P \triangleright \Sigma \cdot k : S \overset{T-Send-u}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k\,?(U); P \triangleright \Sigma \cdot k :?(U); S$

Case (T-Send-s) Let $\Theta; \Gamma \vdash k\,!\langle k' \rangle; P \triangleright \Sigma \cdot k :!\langle S' \rangle; S \cdot k' : S'$ then by (T-Send-s) we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot k : S \overset{T-Send-s}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k\,!\langle k' \rangle; P \triangleright \Sigma \cdot k :!\langle S' \rangle; S \cdot k' : S'$

Case (T-Receive-s) Let $\Theta; \Gamma \vdash k\,?(x); P \triangleright \Sigma \cdot k :?(S'); S$ then by (T-Receive-u) we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S \cdot x : S' \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot k : S \cdot x : S' \overset{T-Receive-s}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k\,?(x); P \triangleright \Sigma \cdot k :?(S'); S$

Case (T-Select) Let $\Theta; \Gamma \vdash k \triangleleft l_i; P \triangleright \Sigma \cdot k : \oplus\{l_i : S_i\}_{i \in I}$ then by (T-Select) we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Sigma \cdot k : S_i \overset{T-Select}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k \triangleleft l_i; P \triangleright \Sigma \cdot k : \oplus\{l_i : S_i\}_{i \in I}$

Case (T-Branch) Let $\Theta; \Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : \&\{l_i : S_i\}_{i \in I}$ then by (T-Branch) we have that $\forall i \in I, \Theta; \Gamma \vdash P_i \triangleright \Sigma \cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash P_i \triangleright \Sigma \cdot k : S_i \overset{T-Branch}{\Longrightarrow} \Theta \cdot \mathbf{X} : T; \Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : \&\{l_i : S_i\}_{i \in I}$

Case (T-If) Let $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$ then by (T-If) we have that $\Gamma \vdash e \triangleright \mathsf{bool}, \Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta\mathbf{X} : T;\Gamma \vdash P \triangleright \Sigma, \Theta\mathbf{X} : T;\Gamma \vdash Q \triangleright \Sigma \overset{T-If}{\Longrightarrow} \Theta\mathbf{X} : T;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$

Case (T-Par) Let $\Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$ then by (T-Par) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma' \overset{I.H.}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash P \triangleright \Sigma, \Theta\cdot\mathbf{X} : T;\Gamma \vdash Q \triangleright \Sigma' \overset{T-Select}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$

Case (T-Restr-a) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ u : \langle S\rangle)\ P \triangleright \Sigma$ then by (T-Restr-a) we have that $\Theta;\Gamma\cdot u : \langle S\rangle \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma\cdot u : \langle S\rangle \vdash P \triangleright \Sigma \overset{T-Restr-a}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash (\boldsymbol{\nu}\ u : \langle S\rangle)\ P \triangleright \Sigma$

Case (T-Def-in) Let $\tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$ then by (T-Def-in) we have that $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma$. If we apply the induction hypothesis on the last two judgements we have that $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S}\cdot\mathbf{X} : T;\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S}\cdot\mathbf{X} : T;\Gamma \vdash Q \triangleright \Sigma \overset{T-Def-in}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$

Case (T-Typecase) Let $\Theta;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$ then by (T-Typecase) we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{T-Branch}{\Longrightarrow} \Theta\cdot\mathbf{X} : T;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$

Part 2

The pattern of the Part 2 proof is very similar to the proof in Part 1.

Basic Step:

Case (T-Nill) Let $u \notin \mathrm{dom}(\Gamma)$ then we conclude that $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma$ by (T-Nill) that Σ completed and trivialy $\Theta;\Gamma\cdot u : U \vdash \mathbf{0} \triangleright \Sigma$

Case (T-Process-var) Trivial. Let $u \notin \mathrm{dom}(\Gamma)$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$ then Σ completed, $\tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Gamma \vdash \tilde{x}_U \triangleright \tilde{U}$ and $\Theta\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot u : U' \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$

Induction Hypothesis:

Assume that $\Theta;\Gamma \vdash P \triangleright \Sigma$ and $u \notin \mathrm{dom}(\Gamma)$ implies $\Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma$

Induction Step:

For all cases assume that $u \notin \mathrm{dom}(\Gamma)$

Case (T-Sub-s) Let $\Theta;\Gamma \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma$

Case (T-Accept) Let $\Theta;\Gamma \vdash u'(x : S).P \triangleright \Sigma$ then by (T-Accept) we have that $\Gamma \vdash u' \triangleright \langle S\rangle$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot x : S \overset{T-Accept}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash u'(x : S).P \triangleright \Sigma$

Case (T-Request) Let $\Theta;\Gamma \vdash \overline{u'}(x : S).P \triangleright \Sigma$ then by (T-Request) we have that $\Gamma \vdash u' \triangleright \langle\overline{S}\rangle$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot x : S \overset{T-Request}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash \overline{u'}(x : S).P \triangleright \Sigma$

Case (T-Send-u) Let $\Theta;\Gamma \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k : !\langle U\rangle; S$ then by (T-Send-u) we have that $\Gamma \vdash e \triangleright U$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot k : S \overset{T-Send-u}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k : !\langle U\rangle; S$

Case (T-Receive-u) Let $\Theta;\Gamma \vdash k\,?(x); P \triangleright \Sigma\cdot k : ?(U); S$ then by (T-Send-u) we have that $\Theta;\Gamma\cdot x : U \vdash P \triangleright \Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot x : U\cdot u : U' \vdash P \triangleright \Sigma\cdot k : S \overset{T-Send-u}{\Longrightarrow} \Theta;\Gamma\cdot u : U' \vdash k\,?(U); P \triangleright \Sigma\cdot k : ?(U); S$

Case (T-Send-s) Let $\Theta;\Gamma \vdash k\,!\langle k'\rangle; P \triangleright \Sigma\cdot k :!\langle S'\rangle; S\cdot k' : S'$ then by (T-Send-s) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot k : S \overset{T-Send-s}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash k\,!\langle k'\rangle; P \triangleright \Sigma\cdot k :!\langle S'\rangle; S\cdot k' : S'$

Case (T-Receive-s) Let $\Theta;\Gamma \vdash k\,?(x); P \triangleright \Sigma\cdot k :?(S'); S$ then by (T-Receive-u) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S\cdot x : S' \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot k : S\cdot x : S' \overset{T-Receive-s}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash k\,?(x); P \triangleright \Sigma\cdot k :?(S'); S$

Case (T-Select) Let $\Theta;\Gamma \vdash k \triangleleft l_i; P \triangleright \Sigma\cdot k : \oplus\{l_i : S_i\}_{i\in I}$ then by (T-Select) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot k : S_i \overset{T-Select}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash k \triangleleft l_i; P \triangleright \Sigma\cdot k : \oplus\{l_i : S_i\}_{i\in I}$

Case (T-Branch) Let $\Theta;\Gamma \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : \&\{l_i : S_i\}_{i\in I}$ then by (T-Branch) we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{T-Branch}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : \&\{l_i : S_i\}_{i\in I}$

Case (T-If) Let $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$ then by (T-If) we have that $\Gamma \vdash e \triangleright \mathsf{bool}, \Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma u : U \vdash P \triangleright \Sigma, \Theta;\Gamma u : U \vdash Q \triangleright \Sigma \overset{T-If}{\Longrightarrow} \Theta;\Gamma u : U \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$

Case (T-Par) Let $\Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$ then by (T-Par) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma' \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma, \Theta;\Gamma\cdot u : U \vdash Q \triangleright \Sigma' \overset{T-Select}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$

Case (T-Restr-a) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ u' : \langle S\rangle)\ P \triangleright \Sigma$ then by (T-Restr-a) we have that $\Theta;\Gamma\cdot u : \langle S\rangle \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u' : \langle S\rangle\cdot u : U \vdash P \triangleright \Sigma \overset{T-Restr-a}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash (\boldsymbol{\nu}\ u' : \langle S\rangle)\ P \triangleright \Sigma$

Case (T-Def-in) Let $\Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$ then by (T-Def-in) we have that $\tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma$. If we apply the induction hypothesis on the last two judgements we have that $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U}\cdot u : U' \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot u : U' \vdash Q \triangleright \Sigma \overset{T-Def-in}{\Longrightarrow} \Theta;\Gamma\cdot u : U' \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$

Case (T-Typecase) Let $\Theta;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$ then by (T-Typecase) we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{T-Branch}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$

Part 3

The pattern of the Part 3 proof is very similar to the proof in Parts 1 and 2.

Basic Step:

Case (T-Nill) Let $k_e \notin \mathrm{dom}(\Sigma)$, $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma$ then , Σ completed, $S_e = \texttt{end}$ or $S_e = \bot$ and trivialy $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma\cdot k_e : S_e$

Case (T-Process-var) Let $k_e \notin \mathrm{dom}(\Sigma), \tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Gamma \vdash \tilde{x}_U \triangleright \tilde{U}$ then $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$, Σ completed, $S_e = \texttt{end}$ or $S_e = \bot$ and $\Theta\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S\cdot k_e : S_e$

Induction Hypothesis:

Assume that $\Theta;\Gamma \vdash P \triangleright \Sigma$ and $k \notin \mathrm{dom}(\Sigma), S_e = \texttt{end}$ or $S_e = \bot$ implies $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k_e : S_e$

Induction Step:

For all cases assume that $k \notin \mathrm{dom}(\Sigma), S_e = \texttt{end}$ or $S_e = \bot$

Case (T-Sub-s) Let $\Theta;\Gamma \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : U \vdash P \triangleright \Sigma\cdot k_e : S_e$

Case (T-Accept) Let $\Theta;\Gamma \vdash u(x : S).P \triangleright \Sigma$ then by (T-Accept) we have that $\Gamma \vdash u \triangleright \langle S\rangle$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S\cdot k_e : S_e \overset{T-Accept}{\Longrightarrow} \Theta;\Gamma \vdash u(x : S).P \triangleright \Sigma\cdot k_e : S_e$

Case (T-Request) Let $\Theta;\Gamma \vdash \overline{u}(x : S).P \triangleright \Sigma$ then by (T-Request) we have that $\Gamma \vdash u \triangleright \langle\overline{S}\rangle$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S\cdot k_e : S_e \overset{T-Request}{\Longrightarrow} \Theta;\Gamma \vdash \overline{u}(x : S).P \triangleright \Sigma\cdot k_e : S_e$

Case (T-Send-u) Let $\Theta;\Gamma \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k :!\langle U\rangle; S$ then by (T-Send-u) we have that $\Gamma \vdash e \triangleright U$ and $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S\cdot k_e : S_e \overset{T-Send-u}{\Longrightarrow} \Theta;\Gamma \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k :!\langle U\rangle; S\cdot k_e : S_e$

Case (T-Receive-u) Let $\Theta;\Gamma \vdash k\,?(x); P \triangleright \Sigma\cdot k :?(U); S$ then by (T-Send-u) we have that $\Theta;\Gamma\cdot x : U \vdash P \triangleright \Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot x : U \vdash P \triangleright \Sigma\cdot k : S\cdot k_e : S_e \overset{T-Send-u}{\Longrightarrow} \Theta;\Gamma \vdash k\,?(U); P \triangleright \Sigma\cdot k :?(U); S\cdot k_e : S_e$

Case (T-Send-s) Let $\Theta;\Gamma \vdash k\,!\langle k'\rangle; P \triangleright \Sigma\cdot k :!\langle S'\rangle; S\cdot k' : S'$ then by (T-Send-s) we have that $\Theta;\Gamma \vdash P\triangleright\Sigma\cdot k : S \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P\triangleright\Sigma\cdot k : S\cdot k_e : S_e \overset{T-Send-s}{\Longrightarrow} \Theta;\Gamma \vdash k\,!\langle k'\rangle; P\triangleright\Sigma\cdot k :!\langle S'\rangle; S\cdot k' : S'\cdot k_e : S_e$

Case (T-Receive-s) Let $\Theta;\Gamma \vdash k\,?(x); P \triangleright \Sigma\cdot k :?(S'); S$ then by (T-Receive-u) we have that $\Theta;\Gamma \vdash P\triangleright\Sigma\cdot k : S\cdot x : S' \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P\triangleright\Sigma\cdot k : S\cdot x : S'\cdot k_e : S_e \overset{T-Receive-s}{\Longrightarrow} \Theta;\Gamma \vdash k\,?(x); P\triangleright\Sigma\cdot k :?(S'); S\cdot k_e : S_e$

Case (T-Select) Let $\Theta;\Gamma \vdash k \triangleleft l_i; P \triangleright \Sigma\cdot k : \oplus\{l_i : S_i\}_{i\in I}$ then by (T-Select) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S_i\cdot k_e : S_e \overset{T-Select}{\Longrightarrow} \Theta;\Gamma \vdash k \triangleleft l_i; P \triangleright \Sigma\cdot k : \oplus\{l_i : S_i\}_{i\in I}\cdot k_e : S_e$

Case (T-Branch) Let $\Theta;\Gamma \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : \&\{l_i : S_i\}_{i\in I}$ then by (T-Branch) we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i\cdot k_e : S_e \overset{T-Branch}{\Longrightarrow} \Theta;\Gamma \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : \&\{l_i : S_i\}_{i\in I}\cdot k_e : S_e$

Case (T-If) Let $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$ then by (T-If) we have that $\Gamma \vdash e \triangleright \mathsf{bool}, \Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q\triangleright\Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P\triangleright\Sigma k_e : S_e, \Theta;\Gamma \vdash Q\triangleright\Sigma k_e : S_e \overset{T-If}{\Longrightarrow} \Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\triangleright\Sigma k_e : S_e$

Case (T-Par) Let $\Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$ then by (T-Par) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma' \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P \triangleright \Sigma\cdot k_e : S_e, \Theta;\Gamma \vdash Q \triangleright \Sigma'\cdot k_e : S_e \overset{T-Select}{\Longrightarrow} \Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'\cdot k_e : S_e$

Case (T-Restr-a) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ u : \langle S\rangle)\ P \triangleright \Sigma$ then by (T-Restr-a) we have that $\Theta;\Gamma\cdot u : \langle S\rangle \vdash P \triangleright \Sigma \overset{I.H.}{\Longrightarrow} \Theta;\Gamma\cdot u : \langle S\rangle \vdash P \triangleright \Sigma\cdot k_e : S_e \overset{T-Restr-a}{\Longrightarrow} \Theta;\Gamma \vdash (\boldsymbol{\nu}\ u : \langle S\rangle)\ P \triangleright \Sigma\cdot k_e : S_e$

Case (T-Def-in) Let $\Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$ then by (T-Def-in) we have that $\tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma$. If we apply the induction hypothesis on the last two judgements we have that $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}\cdot k_e : S_e$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma\cdot k_e : S_e \overset{T-Def-in}{\Longrightarrow} \Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma\cdot k_e : S_e$

Case (T-Typecase) Let $\Theta;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$ then by (T-Typecase) we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i \overset{I.H.}{\Longrightarrow} \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i\cdot k_e : S_e \overset{T-Branch}{\Longrightarrow} \Theta;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{l_i : S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}\cdot k_e : S_e$

We see that in all inductive cases of all the parts, we follow an identical pattern for proving the desired conclusion. Let a process P, from the last rule applied (given by the case we are examining) we find the condition typings for P. We apply the induction hypothesis on the typing conditions and then by applying the case rule we complete the case proof.

■

6.2.2 Runtime Weakening Lemma

We extend the weakening lemma to cover the runtime typing rules cases. Formally:

Lemma 6.2.2 *Runtime Weakening*

Assume $\Theta; \Gamma \vdash P \triangleright \Delta$

1. If $\boldsymbol{X} \notin dom(\Theta)$ then $\Theta \cdot \boldsymbol{X} : T; \Gamma \vdash P \triangleright \Delta$
2. If $u \notin dom(\Gamma)$ then $\Theta; \Gamma \cdot u : U \vdash P \triangleright \Delta$
3. If $k \notin dom(\Sigma)$ and $S = \texttt{end}$ or $S = \bot$then $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : S$

The proof requires for induction on the structure of the typing tree. The leaves of the typing tree for process, thus the basic step cases, are always formed by rules (Ses-Config-sess) and (Sh-Config). Rule pairs (T-Par), (Par) and rules (T-Restr-a), (Restr-sess) are essentially the same rule and we do not check twice for these cases. We also consider the *weakening* lemma(6.2.1) for the proof.

Proof.

Part 1

Basic Step:

Case (Sh-Config) Let $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$ and $\mathbf{X} \notin \mathrm{dom}(\Theta)$. By case rule we conclude $\Gamma \vdash a \triangleright \langle S \rangle$ and $\forall i, \Gamma; \Sigma_i \vdash s_i : S_i$. This trivially implies that $\Theta \cdot \mathbf{X} : T; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$

Case (Ses-Config) Let $\Theta; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$ and $\mathbf{X} \notin \mathrm{dom}(\Theta)$. By case rule we conclude $\forall i, \Gamma; \Sigma_i \vdash h_i : \tau_i$. This trivially implies that $\Theta \cdot \mathbf{X} : T; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$

Induction Hypothesis:

Let $\Theta; \Gamma \vdash P \triangleright \Delta$. If $\mathbf{X} \notin \mathrm{dom}(\Theta)$ then $\Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Delta$

Induction Step:

For all cases assume that$\mathbf{X} \notin \mathrm{dom}(\Theta)$.

Case ($\Delta = \Sigma$) Let $\Theta; \Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *weakening* lemma (6.2.1). Note that the weakening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau}_1) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau}_2)$. Apply the induction hypothesis to get $\Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau}_1) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau}_2)$. The result comes by applying the case rule $\Theta \cdot \mathbf{X} : T; \Gamma \vdash P \triangleright \Delta$.

Part 2

Basic Step:

Case (Ses-Config) Let $u \notin \mathrm{dom}(\Gamma)$ and $\Theta; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$. By case rule we conclude $\forall i, \Gamma; \Sigma_i \vdash h_i : \tau_i$. Clearly $\forall i, \Gamma u : U; \Sigma_i \vdash h_i : \tau_i$. This implies that $\Theta; \Gamma u : U \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$

Case (Sh-Config) Let $u \notin \mathrm{dom}(\Gamma)$ and $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$. By case rule we conclude $\Gamma \vdash a \triangleright \langle S \rangle$ and $\forall i, \Gamma; \Sigma_i \vdash s_i : S_i$. Clearly $\forall i, \Gamma \cdot u : U; \Sigma_i \vdash s_i : S_i$. This implies that

$\Theta; \Gamma \cdot u : U \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$

Induction Hypothesis:

Let $\Theta; \Gamma \vdash P \triangleright \Delta$. If $u \notin \mathrm{dom}(\Gamma)$ then $\Theta; \Gamma \cdot u : U \vdash P \triangleright \Delta$

Induction Step:

For all cases assume that$u \notin \mathrm{dom}(\Gamma)$.

Case ($\Delta = \Sigma$) Let $\Theta; \Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *weakening* lemma (6.2.1). Note that the weakening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2})$. Apply the induction hypothesis to get $\Theta; \Gamma \cdot u : U \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2})$. The result comes by applying the case rule $\Theta; \Gamma \cdot u : U \vdash P \triangleright \Delta$.

Part 3

Basic Step:

Case (Ses-Config) Let $k_e \notin \mathrm{dom}(\Sigma)$ and $\Theta; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$. By case rule we conclude $\forall i, \Gamma; \Sigma_i \vdash h_i : \tau_i$. This trivially implies that $\Theta; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau} \cdot k_e : S_e$

Case (Sh-Config) Let $k_e \notin \mathrm{dom}(\Sigma)$ and $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$. By case rule we conclude $\Gamma \vdash a \triangleright \langle S \rangle$ and $\forall i, \Gamma; \Sigma_i \vdash s_i : S_i$. This trivially implies that $\Theta; \Gamma \cdot u : U \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a \cdot k_e : S_e$

Induction Hypothesis:

Let $\Theta; \Gamma \vdash P \triangleright \Delta$. If $k_e \notin \mathrm{dom}(\Delta)$ then $\Theta; \Gamma \cdot u : U \vdash P \triangleright \Delta \cdot k_e : S_e$.

Induction Step:

For all cases assume that$k_e \notin \mathrm{dom}(\Sigma)$.

Case ($\Delta = \Sigma$) Let $\Theta; \Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *weakening* lemma (6.2.1). Note that the weakening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2})$. Apply the induction hypothesis to get $\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : (S_1, [S_1]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2}) \cdot k_e : S_e$. The result comes by applying the case rule $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k_e : S_e$.

■

6.2.3 Strengthening Lemma

The strengthening lemma is the opposite of the weakening lemma. If we have a process judgement then *strengthen* the typing mappings of the judgement, in a sense that we can arbitrary remove information under conditions, in the mapping without changing the typing of a whole process. Formally:

Lemma 6.2.3 *Strengthening*

Assume $\Theta; \Gamma \vdash P \triangleright \Sigma$

1. If $\boldsymbol{X} \notin \mathit{fpv}(P)$ then $\Theta \backslash \boldsymbol{X}; \Gamma \vdash P \triangleright \Sigma$

2. If $u \notin \mathit{fn}(P)$ then $\Theta; \Gamma \backslash u \vdash P \triangleright \Sigma$

3. If $k \notin \mathit{fs}(P)$ then $\Theta; \Gamma \vdash P \triangleright \Sigma \backslash k$

The proof follows a simple induction on the static typing structure of process P by using the last rule applied on P. The basic step is given by the leaves of the tree which are rules (T-Nill) and (T-Process-var).

Proof.

Part 1

Basic Step:

Case (T-Nill) It is trivial that $\mathbf{X} \notin \mathrm{fpv}(\mathbf{0})$. $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$ trivially implies $\Theta \backslash \mathbf{X}; \Gamma \vdash \mathbf{0} \triangleright \Sigma$

Case (T-Process-var) It is trivial that $\mathbf{X} \notin \mathrm{fpv}(\mathbf{Y}\langle \tilde{x} \rangle)$. $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash \mathbf{Y}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{x}_S$. This trivially implies $(\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}) \backslash \mathbf{X}; \Gamma \vdash \mathbf{Y}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{x}_S$

Induction Hypothesis:

Assume that $\Theta; \Gamma \vdash P \triangleright \Sigma$ and $\mathbf{X} \notin \mathrm{fpv}(P)$ implies $\Theta \backslash \mathbf{X}; \Gamma \vdash P \triangleright \Sigma$
Induction Step:

For all cases assume that $\mathbf{X} \notin \mathrm{fpv}(P)$ and $\mathbf{X} \notin \mathrm{fpv}(Q)$ whenever it applies.

Case (T-Sub) $\Theta; \Gamma \vdash P \triangleright \Sigma \stackrel{I.H}{\Longrightarrow} \Theta \backslash \mathbf{X}; \Gamma \vdash P \triangleright \Sigma$

We group the next two cases

Case (T-Accept) u is in input state $u(x)$
Case (T-Request) u is in output state $\overline{u}(x)$

Assume that $\Theta; \Gamma \vdash u(x). \triangleright \Sigma$ then by last rule applied we have that $\Gamma \vdash u \triangleright \langle S \rangle, \Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S$ and by the induction hypothesis we have $\Theta \backslash \mathbf{X}; \Gamma \vdash P \triangleright \Sigma \cdot x : S$ by applying to the last judgement the case rule we have that $\Theta \backslash \mathbf{X}; \Gamma \vdash u(x).P \triangleright \Sigma$ completing the above cases.

For the next five cases consider a more relaxed definition of contexts, with context $C = -; P$. Symbol p denotes the prefix of process P.

Case (T-Send-u) $p = k\,!\langle e \rangle, \Sigma_1 = k : S, \Sigma_2 = k :!\langle U \rangle; S$
Case (T-Receive-u) $p = k\,?(x), \Sigma_1 = k : S, \Sigma_2 = k :?(U); S$
Case (T-Send-s) $p = k\,!\langle k' \rangle, \Sigma_1 = k : S, \Sigma_2 = k :?(S'); S \cdot k' : S'$
Case (T-Receive-s) $p = k\,?(x), \Sigma_1 = k : S \cdot x : S', \Sigma_2 = k :?(S'); S$
Case (T-Select) $p = k \triangleleft l_i, \Sigma_1 = k : S_i, \Sigma_2 = k : \oplus\{l_i : S_i\}_{i \in I}$

Assume that $\Theta; \Gamma \vdash C[p] \triangleright \Sigma \cdot \Sigma_1$ then by last rule applied we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot \Sigma_2$ and by the induction hypothesis we have $\Theta \backslash \mathbf{X}; \Gamma \vdash P \triangleright \Sigma \cdot \Sigma_1$ by applying to the last judgement the case rule we have that $\Theta \backslash \mathbf{X}; \Gamma \vdash C[p] \triangleright \Sigma \cdot \Sigma_2$ completing the above cases.

Case (T-Branch) Assume that $\Theta; \Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Sigma \cdot k : \&\{l_i : S_i\}_{i \in I}$ then by last rule applied we have that $\forall i \in I, \Theta; \Gamma \vdash P_i \triangleright \Sigma k : S_i$ and by the induction hypothesis we have $\Theta \backslash \mathbf{X}; \Gamma \vdash P_i \triangleright \Sigma k : S_i$ by

applying to the last judgement the case rule we have that $\Theta\backslash\mathbf{X};\Gamma \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma k : \&\{l_i : S_i\}_{i\in I}$

Case (T-If) Assume that $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$ then by last rule applied we have that $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma$ and by the induction hypothesis we have $\Theta\backslash\mathbf{X};\Gamma \vdash P \triangleright \Sigma, \Theta\backslash\mathbf{X};\Gamma \vdash Q \triangleright \Sigma$ by applying to the last two judgements the case rule we have that $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma$

Case (T-Par) Assume that $\Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\cdot\Sigma'$ then by last rule applied we have that $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma'$ and by the induction hypothesis we have $\Theta\backslash\mathbf{X};\Gamma \vdash P \triangleright \Sigma, \Theta\backslash\mathbf{X};\Gamma \vdash Q \triangleright \Sigma'$ by applying to the last two judgements the case rule we have that $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma\cdot\Sigma'$

Case (T-Restr-a) Assume that $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ u)\ P \triangleright \Sigma$ then by last rule applied we have that $\Gamma \vdash u : \triangleright\langle S\rangle, \Theta;\Gamma \vdash P \triangleright \Sigma$, easy to apply the induction hypothesis to get $\Theta\backslash\mathbf{X};\Gamma \vdash P \triangleright \Sigma$ and by applying (T-Restr-a) we get $\Theta\backslash\mathbf{X};\Gamma \vdash (\boldsymbol{\nu}\ u)\ P \triangleright \Sigma$

Case (T-Def-in) Assume $\Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$ then by (T-Def-in) we have that $\tilde{x} = \tilde{x}_U\cdot\tilde{x}_S, \Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma$. If we apply the induction hypothesis on the last two judgements we have that $(\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S})\backslash\mathbf{X};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $(\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S})\backslash\mathbf{X};\Gamma \vdash Q \triangleright \Sigma \overset{T-Def-in}{\Longrightarrow} \Theta\backslash\mathbf{X};\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$

Case (T-Typecase) Assume that $\Theta;\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$ then by last rule applied we have that $\forall i \in I, \Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i$ and by the induction hypothesis we have $\Theta\backslash\mathbf{X};\Gamma \vdash P_i \triangleright \Sigma\cdot k : S_i$ by applying to the last judgement the case rule we have that $\Theta\backslash\mathbf{X};\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : [S_i]_{i\in I}$

Part 2

Basic Step:

Case (T-Nill) Trivially $u \notin \mathrm{fn}(\mathbf{0})$. $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma$ trivialy implies Σ completed and $\Theta;\Gamma\backslash u \vdash \mathbf{0} \triangleright \Sigma$

Case (T-Process-var) Trivialy $u \notin \mathrm{fn}(\mathbf{X}\langle\tilde{x}\rangle)$. $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$ trivially implies $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\backslash u \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$

Induction Hypothesis:

Assume that if $u \notin \mathrm{fn}(\mathbf{X}\langle\tilde{x}\rangle)$ and $\Theta;\Gamma \vdash P \triangleright \Sigma$ then $\Theta;\Gamma\backslash u \vdash \mathbf{0} \triangleright \Sigma$

Induction Step:

For all cases consider that $u \notin \mathrm{fn}(P)$ and $u \notin \mathrm{fn}(Q)$ whenever it applies.

Case (T-Sub) This is trivial since process P can be typed directly from the induction hypothesis.

Case (T-Accept) Let $\Theta;\Gamma \vdash u'(x).P \triangleright \Sigma$. $u \neq u'$ because we want $u \notin \mathrm{fn}(u'(x).P)$. So by (T-Accept) we have that $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S$ and $\Gamma \vdash u' \triangleright \langle S\rangle$. An application of the induction hypothesis will give us $\Theta;\Gamma\backslash u \vdash P \triangleright \Sigma\cdot x : S$. Rule (T-Accept) will give us the desired result $\Theta;\Gamma\backslash u \vdash u'(x).P \triangleright \Sigma$

Case (T-Request) Similar arguments to case (T-Accept).

Case (T-Send-u) Let $\Theta;\Gamma \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k :!\langle U\rangle; S$. (T-Send-u) implies $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot x : S$ and $\Gamma \vdash e \triangleright U$. The induction hypothesis implies $\Theta;\Gamma\backslash u \vdash P \triangleright \Sigma\cdot x : S$. Rule (T-Send-u) will give us the desired result $\Theta;\Gamma\backslash u \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k :!\langle U\rangle; S$

Case (T-Receive-u), (T-Send-s), (T-Receive-s) and (T-Select) follow the same argumentation as (T-Send-u). On the proof someone must be careful to give proper typings and other conditions between various rule applications.

Case (T-Branch) Let $\Theta;\Gamma \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma k : \&\{l_i : S_i\}_{i\in I}$. By (T-Branch) $\forall i \in I \Theta;\Gamma \vdash P_i \triangleright \Sigma k : S_i$. Apply the induction hypothesis on all processes, so $\forall i \in I \Theta;\Gamma\backslash u \vdash P_i \triangleright \Sigma\cdot k : S_i$. Apply (T-Branch), so $\Theta;\Gamma\backslash u \vdash k \triangleright \{l_i : P_i\}_{i\in I} \triangleright \Sigma\cdot k : \&\{l_i : S_i\}_{i\in I}$

Case (T-Par) Let $\Theta;\Gamma \vdash P \mid Q \triangleright \Sigma\Sigma'$. By the case rule we have $\Theta;\Gamma \vdash P \triangleright \Sigma, \Theta;\Gamma \vdash Q \triangleright \Sigma'$. The induction hypothesis gives $\Theta;\Gamma\backslash u \vdash P \triangleright \Sigma, \Theta;\Gamma\backslash u \vdash Q \triangleright \Sigma'$ so the case rule will give us $\Theta;\Gamma\backslash u \vdash P \mid Q \triangleright \Sigma\Sigma'$

Case (T-Restr-a) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, u')\ P \triangleright \Sigma$. By case rule we have $\Theta;\Gamma\cdot u' : \langle S\rangle \vdash P \triangleright \Sigma$ and $u \neq u'$ because $u \notin \mathrm{fn}(P)$ and u' free in P. Then by induction hypothesis $\Theta;\Gamma\cdot u' : \langle S\rangle\backslash u \vdash P \triangleright \Sigma$. The result comes by applying (T-Restr-a) to get $\Theta;\Gamma\backslash u \vdash (\boldsymbol{\nu}\, u')\ P \triangleright \Sigma$

Case (T-if) and (T-Def-in) are similar to case (T-Par). Be careful with the process typing when concluding from rules.

Case (T-Typecase) Similar to case (T-Branch).

Part 3

Similar to the other 2 parts.

Basic Step:

Case (T-Nill) Assume $\Theta;\Gamma \vdash \mathbf{0}\Sigma$. It is obvious that $s \notin \mathrm{fs}(\mathbf{0})$ then is trivial to conclude that Σ completed, $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma\backslash s$

Case (T-Process-var) Trivially $s \notin \mathrm{fs}(\mathbf{X}\langle\tilde{x}\rangle)$. $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright \Sigma\cdot\tilde{x} : \tilde{x}_S$ trivialy implies $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash \mathbf{Y}\langle\tilde{x}\rangle \triangleright (\Sigma\cdot\tilde{x} : \tilde{x}_S)\backslash s$

Induction Hypothesis:

If $s \notin \mathrm{fs}(P)$ and $\Theta;\Gamma \vdash P \triangleright \Sigma$ then $\Theta;\Gamma \vdash P \triangleright \Sigma\backslash s$

Induction Step:

Consider that in all cases $s \notin \mathrm{fs}(P)$ and $s \notin \mathrm{fs}(Q)$.

Case (T-Sub) Straightforward from the induction hypothesis.

Case (T-Accept) Let $\Theta;\Gamma \vdash u(s').P \triangleright \Sigma$. Then by case rule we have that $\Gamma \vdash s' \triangleright \langle S\rangle, \Theta;\Gamma \vdash P \triangleright \Sigma s' : S$. Apply the induction hypothesis to get $\Theta;\Gamma \vdash P \triangleright \Sigma\cdot s' : S\backslash s$. The result comes by applying the (T-Accept) rule to get $\Theta;\Gamma \vdash u(s').P \triangleright \Sigma\backslash s$

Case (T-Request) is similar to case (T-Accept).

Case (T-Send-u) Assume $\Theta;\Gamma \vdash k\,!\langle e\rangle; P \triangleright \Sigma\cdot k :!\langle U\rangle; S$ then by (T-Send-u) we conclude $\Gamma \vdash e \triangleright U, \Theta;\Gamma \vdash P \triangleright \Sigma\cdot k : S$. Clearly $s \neq k$ because $s \notin \mathrm{fs}(P)$ and the induction hypothesis implies $\Theta;\Gamma \vdash P \triangleright (\Sigma\cdot k : S)\backslash s$.

Case (T-Receive-u), (T-Send-s), (T-Receive-s), (T-Select) and (T-Restr-a) follow similar argumentation as (T-Send-u)). The difference lies in the conclusion of the typings, but in all cases all typings are straightforward to be obtained.

Case (T-Branch), (T-Typecase) follows the same argumentation as (T-Send-u). The diference is that we may have more than one processes to type after the implication from the case rule, but this is again straightforward.
Case (T-If), (T-Par), (T-Def-in) follow the similar pattern as (T-Send-u). Here though the case rule implies the typing of two processes. The induction hypothesis is applied to the two processes. The rest are easy and straightforward.

■

6.2.4 Runtime Stengthening Lemma

We extend the strengthening lemma to cover the runtime typing rules cases. Formally:

Lemma 6.2.4 *Runtime Strengthening*

Assume $\Theta; \Gamma \vdash P \triangleright \Delta$

1. If $\boldsymbol{X} \notin \mathit{fpv}(P)$ then $\Theta \backslash \boldsymbol{X}; \Gamma \vdash P \triangleright \Delta$
2. If $u \notin \mathit{fn}(P)$ then $\Theta; \Gamma \backslash u \vdash P \triangleright \Delta$
3. If $k \notin \mathit{fs}(P)$ then $\Theta; \Gamma \vdash P \triangleright \Delta \backslash k$

The proof requires for induction on the structure of the typing tree. The leaves of the typing tree for process, thus the basic step cases, are always formed by rules (Ses-Config-sess) and (Sh-Config). Ruke rule pairs (T-Par), (Par) and rules (T-Restr-a), (Restr-sess) are essentially the same rule and we do not check twice for these cases. We also consider the *strengthening* lemma(6.2.3) for the proof.

Proof.

Part 1

Basic Step:

Case (Ses-Config) Let $\Theta; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$ and it is clear that $\mathbf{X} \notin \mathrm{fpv}(s\,[S] : \vec{h_i})$. This trivially implies that $\Theta \backslash \mathbf{X}; \Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$

Case (Sh-Config) Let $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : \vec{[S_i]}\,\varepsilon \odot a$ and it is clear that $\mathbf{X} \notin \mathrm{fpv}(a\,[\langle S \rangle] : \vec{s_i})$. This trivially implies that $\Theta \backslash \mathbf{X}; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : \vec{[S_i]}\,\varepsilon \odot a$

Induction Hypothesis:

Let $\Theta; \Gamma \vdash P \triangleright \Delta$. If $\mathbf{X} \notin \mathrm{fpv}(P)$ then $\Theta \backslash \mathbf{X}; \Gamma \vdash P \triangleright \Delta$.

Induction Step:

For all cases assume that $\mathbf{X} \notin \mathrm{fpv}(P)$.

Case ($\Delta = \Sigma$) Let $\Theta; \Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *strengthening* lemma (6.2.3). Note that the strengthening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta;\Gamma \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)$. Apply the induction hypothesis to get $\Theta\backslash\mathbf{X};\Gamma \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)$. The result comes by applying the case rule $\Theta\backslash\mathbf{X};\Gamma \vdash P \triangleright \Delta$.

Part 2

Basic Step:

Case (Ses-Config) Trivially $u \notin \mathrm{fn}(s\,[S] : \vec{h_i})$ and $\Theta;\Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$. By case rule we conclude $\forall i,\Gamma;\Sigma_i \vdash h_i : \tau_i$. Clearly $\forall i,\Gamma\backslash u;\Sigma_i \vdash h_i : \tau_i$. This implies that $\Theta;\Gamma\backslash u \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$

Case (Sh-Config) Trivially $u \notin \mathrm{fn}(a\,[\langle S\rangle] : \vec{s_i})$ and $\Theta;\Gamma \vdash a\,[\langle S\rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$. By case rule we conclude $\Gamma \vdash a \triangleright \langle S\rangle$ and $\forall i,\Gamma;\Sigma_i \vdash s_i : S_i$. Clearly $\forall i,\Gamma\backslash u;\Sigma_i \vdash s_i : S_i$. This implies that $\Theta;\Gamma\backslash u \vdash a\,[\langle S\rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$
Induction Hypothesis:

Let $\Theta;\Gamma \vdash P \triangleright \Delta$. If $u \notin \mathrm{fn}(P)$ *then* $\Theta;\Gamma\backslash u \vdash P \triangleright \Delta$.

Induction Step:

For all cases assume that$u \notin \mathrm{fn}(s\,[S] : \vec{h_i})$.

Case ($\Delta = \Sigma$) Let $\Theta;\Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *strengthening* lemma (6.2.3). Note that the strengthening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta;\Gamma \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)$. Apply the induction hypothesis to get $\Theta;\Gamma\backslash u \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)$. The result comes by applying the case rule $\Theta;\Gamma\backslash u \vdash P \triangleright \Delta$.

Part 3

Basic Step:

Case (Ses-Config) Let $k_e \notin \mathrm{fn}(s\,[S] : \vec{h_i})$ and $\Theta;\Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}$. This trivially implies that $\Theta;\Gamma \vdash s\,[S] : \vec{h_i} \triangleright \vec{\Sigma_i} \odot s : [S]\,\vec{\tau}\backslash k_e$

Case (Sh-Config) Let $k_e \notin \mathrm{dom}(a\,[\langle S\rangle] : \vec{s_i})$ and $\Theta;\Gamma \vdash a\,[\langle S\rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a$. This trivially implies that $\Theta;\Gamma\cdot u : U \vdash a\,[\langle S\rangle] : \vec{s_i} \triangleright \vec{\Sigma_i} \odot s_i : [\vec{S_i}]\,\varepsilon \odot a\backslash k_e$

Induction Hypothesis:

Let $\Theta;\Gamma \vdash P \triangleright \Delta$. $k \notin \mathrm{fs}(P)$ *then* $\Theta;\Gamma \vdash P \triangleright \Delta\backslash k$.

Induction Step:

For all cases assume that $k_e \notin \mathrm{fn}(s\,[S] : \vec{h_i})$.

Case ($\Delta = \Sigma$) Let $\Theta;\Gamma \vdash P \triangleright \Delta$ and $\Delta = \Sigma$. The result comes by application of the *strengthening* lemma (6.2.3). Note that the strengthening lemma uses the induction hypothesis.

Case (Restr-sess) Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ s)\ P \triangleright \Delta$. By case rule we have $\Theta;\Gamma \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)$. Apply the induction hypothesis to get $\Theta;\Gamma \vdash P \triangleright \Delta\cdot s : (S_1,[S_1]\,\vec{\tau}_1)\cdot\overline{s} : (S_2,[S_2]\,\vec{\tau}_2)\backslash k_e$:

S_e. The result comes by applying the case rule $\Theta; \Gamma \vdash P \triangleright \Delta \backslash k_e : S_e$.

■

6.2.5 Substitution Lemma

The substitution lemma states that, under conditions, we can substitute constructs in a process and preserve the process typing. Formally:

Lemma 6.2.5 *Substitution*

1. If $\Gamma \cdot x : U \vdash e \triangleright U'$ and $\Gamma \vdash v \triangleright U$ then $\Gamma \vdash e\{v/x\} \triangleright U'$
2. If $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma$ and $\Gamma \vdash v \triangleright U$ then $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma$
3. If $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S$ then $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot s : S$

The proof is done by induction on the derivation tree that comes up from the typing system. The leaves of the tree are formed by the axiom rules and the typing rules that don't have as condition a process typing (rules (T-Nill), (T-Process-var)). For part 1 we consider only rules that type expressions. For parts 2 and 3 we consider rules that type processes.

Part 1

Basic Step:

Case (T-Var-u) $\Gamma \cdot x : U \vdash x \triangleright U'$ and $\Gamma \vdash v \triangleright U$ then $x\{v/x\} = v$ and it is trivial that $\Gamma \vdash x\{v/x\} \triangleright U'$

Case (T-bool-true), (T-bool-false), (T-Nat), (T-Session-arrived), (T-Shared-Arrived) are trivial to obtain since substitution has no effect.

Case (T-Shared-chan) Two cases exist:

1. $u \neq x$. Trivial because substitution has no effect
2. $u = x$. Same case as (T-Var-u)

Induction Hypothesis:

Assume that if $\Gamma \cdot x : U \vdash e \triangleright U'$ and $\Gamma \vdash v \triangleright U$ then $\Gamma \vdash e\{v/x\} \triangleright U'$

Induction Step:

Case (T-Sub-u) Straightforward by the induction hypothesis.

Case (T-Not) Assume $\Gamma \vdash v \triangleright U, \Gamma \cdot x : U \vdash not(e) \triangleright bool$ then by the case rule we have that $\Gamma \cdot x : U \vdash e \triangleright bool$. An application of the induction hypothesis will give us $\Gamma \vdash e\{v/x\} \triangleright U'$. By applying rule (T-Not) we have $\Gamma x : U \vdash not(e\{v/x\}) \triangleright bool$ which is equivalent to $\Gamma x : U \vdash not(e)\{v/x\} \triangleright bool$

Part 2

Basic Step:

Case (T-Nill) Assume $\Theta; \Gamma \cdot x : U \vdash \mathbf{0} \triangleright \Sigma$ and $\Gamma \vdash v \triangleright U$. The result is trivial because substitution leaves the process intact. $\mathbf{0}\{v/x\} = \mathbf{0}, \Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$

Case (T-Process-var) Assume $\Theta{\cdot}\mathbf{X} : \tilde{U}{\cdot}\tilde{S}; \Gamma{\cdot}x' : U' \vdash \mathbf{X}\langle\tilde{x}\rangle \triangleright \Sigma{\cdot}\tilde{x} : \tilde{S}$ and $\Gamma \vdash v \triangleright U'$. By rule (T-Process-Var) we conclude that Σ completed, $\tilde{x} = \tilde{x}_U \cdot \tilde{x}_S, \Gamma \cdot x' : U' \vdash \tilde{x}_U \triangleright \tilde{U}$ then by Part 1 of *Substitution* we have that $\Gamma \vdash \tilde{x}_U\{v/x'\} \triangleright \tilde{U}$ and by (T-Process-var) we have that $\Theta\mathbf{X} : \tilde{U}{\cdot}\tilde{S}; \Gamma \vdash \mathbf{X}\langle\tilde{x}\rangle\{v/x'\} \triangleright \Sigma{\cdot}\tilde{x} : \tilde{S}$

Induction Hypothesis:

If $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma$ and $\Gamma \vdash v \triangleright U$ then $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma$

Induction Step:

For all cases suppose that $\Gamma \vdash v \triangleright U$.

Case (T-Sub-s) Straightforward from the induction hypothesis.

Case (T-Accept) Assume $\Theta; \Gamma \cdot x : U \vdash u(y).P \triangleright \Sigma$, then two cases exist:

1. $x \neq u$. Then by case rule we conclude that $\Gamma \vdash u \triangleright \langle S\rangle, \Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma \cdot y : S$ and by induction hypothesis we have that $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma{\cdot}y : S$. Finaly we apply (T-Accept) to get $\Theta; \Gamma \vdash u(y).P\{v/x\} \triangleright \Sigma$.
2. $x = u$. Then $(u(y).P)\{v/x\} = v(y).P\{v/x\}$. v is a shared channel and $\Gamma \vdash v \triangleright U$ with $U = \langle S\rangle$. By case rule we conclude $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Sigma \cdot y : S$. Apply the induction hypothesis to get $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma \cdot y : S$. The result comes by applying (T-Accept) and we have that $\Theta; \Gamma \vdash v(y).P\{v/x\} \triangleright \Sigma$.

Case (T-Request) Similar to case (T-Accept).

Case (T-Send-u) Assume $\Theta; \Gamma{\cdot}x : U \vdash k\,!\langle e\rangle; P \triangleright \Sigma{\cdot}k : !\langle U'\rangle; S$. By case rule we have $\Gamma{\cdot}x : U \vdash e \triangleright U', \Theta; \Gamma{\cdot}x : U \vdash P \triangleright \Sigma{\cdot}k : S$. By part 1 of substitution we have that $\Gamma \vdash e\{v/x\} \triangleright U'$, and by induction hypothesis we get $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma{\cdot}k : S$. Apply (T-Send-u) to get $\Theta; \Gamma \vdash k\,!\langle e\{v/x\}\rangle; P\{v/x\} \triangleright \Sigma{\cdot}k : !\langle U'\rangle; S$.

Case (T-Receive-u) Assume $\Theta; \Gamma \cdot x : U \vdash k\,?(y); P \triangleright \Sigma \cdot k : ?(U'); S$. By case rule we get $\Theta; \Gamma \cdot y : U' \cdot x : U \vdash P \triangleright \Sigma \cdot k : S$. Apply induction hypothesis to get $\Theta; \Gamma \cdot y : U' \vdash P\{v/x\} \triangleright \Sigma \cdot k : S$ and by (T-Receive-u) we get $\Theta; \Gamma \vdash k\,?(y); P\{v/x\} \triangleright \Sigma \cdot k : ?(U'); S$

Case (T-Send-s), (T-Receive-s), (T-Select), (T-Branch), (Typecase) are similar to case (T-Receive-u).

Case (T-Par) Assume $\Theta; \Gamma{\cdot}x : U \vdash P \mid Q \triangleright \Sigma\Sigma'$. By (T-Par) we get $\Theta; \Gamma{\cdot}x : U \vdash P \triangleright \Sigma, \Theta; \Gamma{\cdot}x : U \vdash Q \triangleright \Sigma'$ and by induction hypothesis we get $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Sigma, \Theta; \Gamma \vdash Q\{v/x\} \triangleright \Sigma'$. Conclude the result $\Theta; \Gamma \cdot x : U \vdash P\{v/x\} \mid Q\{v/x\} \triangleright \Sigma \cdot \Sigma'$ by applying the case rule.

Case (T-If) Similar to case (T-Par).

Case (T-Restr-a) Assume $\Theta; \Gamma \cdot x : U \vdash (\boldsymbol{\nu}\, u)\ P \triangleright \Sigma$. Two cases exist:

1. $x \neq u$. Similar to case (T-Receive-u).
2. $x = u$. By (T-Restr-a) we have that $\Theta; \Gamma \cdot x : U \cdot u : U \vdash P \triangleright \Sigma$ and by the initial hypothesis we have $\Gamma \vdash v \triangleright U$, by the induction hypothesis we have $\Theta; \Gamma \cdot x : U \cdot u : U \vdash P\{v/x\} \triangleright \Sigma$. By applying (T-Restr-a) we get the required result $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, v)\ P\{v/x\} \triangleright \Sigma$

Case (T-Def-in) Assume $\Theta; \Gamma \cdot x' : U \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma$. Two cases exist:

1. $x' \neq y, \forall y \in \tilde{x}_U$. By (T-Def-in) we have that $\tilde{x} = \tilde{x}_U{\cdot}\tilde{x}_S, \Theta{\cdot}\mathbf{Y} : \tilde{U}{\cdot}\tilde{S}; \Gamma{\cdot}\tilde{x}_U : \tilde{U}{\cdot}x' : U \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot x' : U \vdash Q \triangleright \Sigma$. If we apply the induction hypothesis on the last two

judgements we have that $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot \tilde{x}_U : \tilde{U} \vdash P\{v/x'\} \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash Q\{v/x'\} \triangleright \Sigma \overset{T-Def-in}{\Longrightarrow} \Theta; \Gamma \vdash \texttt{def } \mathbf{Y}\langle \tilde{x} \rangle = P\{vx'/\ \}\texttt{in } Q\{v/x'\} \triangleright \Sigma$

2. $x' = y, \exists y \in \tilde{x}_U$. By (T-Def-in) we have that $\tilde{x} = \tilde{x}_U \cdot \tilde{x}_S, \Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot \tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot x' : U \vdash Q \triangleright \Sigma$ also note that $\Gamma \vdash \tilde{x}\{v/x'\} \triangleright U$ from part 1 of *substitution*. If we apply the induction hypothesis on the last two judgements we have that $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot \tilde{x_U} : \tilde{U} \vdash P\{v/x'\} \triangleright \tilde{x}_S : \tilde{S}$ and $\Theta \cdot \mathbf{Y} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash Q\{v/x'\} \triangleright \Sigma \overset{T-Def-in}{\Longrightarrow} \Theta; \Gamma \vdash \texttt{def } \mathbf{Y}\langle \tilde{x} \rangle = P\{vx'/\ \}\texttt{in } Q\{v/x'\} \triangleright \Sigma$

Part 3

Basic Step:

Case (T-Nill) Let $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma \cdot x : S$. Since $x, s \notin \text{fs}(\mathbf{0})$ we apply strengthening and then weakening to obtain $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma \cdot s : S$. Since substitution on $\mathbf{0}$ process has no effect we can derive $\Theta; \Gamma \vdash \mathbf{0}\{s/x\} \triangleright \Sigma \cdot s : S$ as required.

Case (T-Process-var) Let $\Theta \cdot \mathbf{X} : \tilde{U} \cdot \tilde{S}; \Gamma \cdot x' : U' \vdash \mathbf{X}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{S} \cdot x' : S'$. Assume that $x', s \notin \text{fs}(\mathbf{X}\langle \tilde{x} \rangle)$ then we apply strengthening and then weakening to obtain $\Theta \cdot \mathbf{X} : \tilde{U} \cdot \tilde{S}; \Gamma \vdash \mathbf{X}\langle \tilde{x} \rangle \triangleright \Sigma \cdot \tilde{x} : \tilde{S} \cdot s : S$.

Induction Hypothesis:

If $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S$ then $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot s : S$

Induction Step:

Case (T-Sub-s) Straightforward from the induction hypothesis.

Case (T-Send-u) We have two cases:

1. $x \neq k$. Let $\Theta; \Gamma \vdash k\,!\langle e \rangle; P \triangleright \Sigma \cdot k :!\langle U \rangle; S \cdot x : S'$. From the case rule we have $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S \cdot x : S'$ and $\Gamma \vdash e \triangleright U$. Apply the induction hypothesis to get $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot k : S \cdot s : S'$. By the (T-Send-u) we conclude $\Theta; \Gamma \vdash k\,!\langle e \rangle; P\{s/x\} \triangleright \Sigma \cdot s :!\langle U \rangle; S \cdot s : S'$

2. $x = k$. From the case rule we have $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S$ and $\Gamma \vdash e \triangleright U$. Apply the induction hypothesis to get $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Sigma s : S$. By the (T-Send-u) we conclude $\Theta; \Gamma \vdash s\,!\langle e \rangle; P\{s/x\} \triangleright \Sigma \cdot s :!\langle U \rangle; S$

Case (T-Receive-u), (T-Select), (T-Branch) and (T-Typecase) are similar to case (T-Send-u).

Case (T-Send-s) Three cases exist

1. $x \neq k or x = k, x \neq k'$. Similar to case (T-Send-u)

2. $x \neq k, x = k'$. Let $\Theta; \Gamma \vdash k\,!\langle k' \rangle; P \triangleright \Sigma \cdot k :!\langle S' \rangle; S \cdot k' : S'$. By case rule we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot k : S$, so this case does not apply because we do not have k' typing in process P typing.

Case (T-Receive-s) Similar to case (T-Send-s)

Case (T-Par) Consider $\Theta; \Gamma \vdash P \mid Q \triangleright \Sigma \cdot \Sigma' \cdot x : S$. By case rule we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot x : S, \Theta; \Gamma \vdash Q \triangleright \Sigma'$ or $\Theta; \Gamma \vdash P \triangleright \Sigma, \Theta; \Gamma \vdash Q \triangleright \Sigma' \cdot x : S$. We will only show one of the two cases, since the other case is the same. Apply induction hypothesis and substitution to get $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot s : S, \Theta; \Gamma \vdash Q\{s/x\} \triangleright \Sigma'$. By applying the case rule we get the desired result $\Theta; \Gamma \vdash P \mid Q \triangleright \Sigma \cdot \Sigma' \cdot s : S$.

Case (T-If) Consider $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma \cdot x : S$. By case rule we have that $\Gamma \vdash e \triangleright \mathsf{bool}, \Theta;\Gamma \vdash P \triangleright \Sigma \cdot x : S, \Theta;\Gamma \vdash Q \triangleright \Sigma' \cdot x : S$. Apply induction hypothesis to get $\Theta;\Gamma \vdash P\{s/x\} \triangleright \Sigma \cdot s : S, \Theta;\Gamma \vdash Q\{s/x\} \triangleright \Sigma \cdot s : S$. By applying the case rule we get the desired result $\Theta;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Sigma \cdot s : S$.

Case (T-Restr) Trivial case by following the same pattern as case (T-If).

Case (T-Def-in) Assume $\Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\ \texttt{in}\ Q \triangleright \Sigma \cdot x' : S'$. By (T-Def-in) we have that $\tilde{x} = \tilde{x}_U\tilde{x}_S, \Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}x' : S'$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q \triangleright \Sigma x' : S'$. If we apply the induction hypothesis on the last two judgements we have that $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma\cdot\tilde{x}_U : \tilde{U} \vdash P\{v/x'\} \triangleright \tilde{x}_S : \tilde{S}\cdot s : S'$ and $\Theta\cdot\mathbf{Y} : \tilde{U}\cdot\tilde{S};\Gamma \vdash Q\{v/x'\} \triangleright \Sigma\cdot s : S' \overset{T-Def-in}{\Longrightarrow} \Theta;\Gamma \vdash \texttt{def}\ \mathbf{Y}\langle\tilde{x}\rangle = P\{vx'/\ \}\texttt{in}\ Q\{v/x'\} \triangleright \Sigma\cdot s : S'$ ■

6.2.6 Runtime Substitution Lemma

The runtime substitution lemma extends the substitution lemma (6.2.5). Formally:

Lemma 6.2.6 *Runtime Substitution*

1. If $\Gamma\cdot x : U;\Sigma \vdash e : U'$ and $\Gamma \vdash v \triangleright U$ then $\Gamma;\Sigma \vdash e\{v/x\} \triangleright U'$
2. If $\Theta;\Gamma\cdot x : U \vdash P \triangleright \Delta$ and $\Gamma \vdash v \triangleright U$ then $\Theta;\Gamma \vdash P\{v/x\} \triangleright \Delta$
3. If $\Theta;\Gamma \vdash P \triangleright \Delta\cdot x : S$ then $\Theta;\Gamma \vdash P\{s/x\} \triangleright \Delta\cdot s : S$

For part 1 we consider all the expression typing judgements.

The proof for parts 1 and 2 is done by induction on the derivation tree that comes up from the typing system. The leaves of the tree are formed by the axiom rules and the typing rules that don't have as condition a process typing. Cases (Par) and (Restr-chan) are almost identical with their static typing correspondence so they are proven again.

Part 1

Case (Message-u) Let $\Gamma\cdot x : U;\Sigma \vdash e : U'$ and $\Gamma \vdash v \triangleright U$. By (Message-u) we have $\Gamma\cdot x : U \vdash e \triangleright U'$. By *substitution* lemma (6.2.5) we conclude $\Gamma \vdash e\{v/x\} \triangleright U'$. Apply the case rule to get $\Gamma;\Sigma \vdash e\{v/x\} : U'$

Case (Message-s) Let $\Gamma\cdot x : U;\Sigma \vdash s : S$ and $\Gamma \vdash v \triangleright U$. Substitution has no effect so the result is trivial to be obtained.

Case (Label) Again the result is trivial to be obtained, because substitution has no effect.

Part 2

Basic Step:

Case (Ses-Config) Let $\Theta;\Gamma\cdot x : U \vdash s\,[S] : \vec{h}_{i\,i\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\vec{\tau}$ and $\Gamma \vdash v \triangleright U$. By rule (Ses-Config) we conclude that $\forall i \in I, \Gamma\cdot x : U;\Sigma_i \vdash h_i : \tau_i$. Two cases apply:

1. $x \neq h_i, \forall i \in I$. By Part 1 of *runtime substitution* we conclude that $\forall i \in I, \Gamma;\Sigma_i \vdash h_i\{v/U\} : \tau_i$. Apply the case rule to get $\Theta;\Gamma \vdash s\,[S] : \vec{h}_{i\,i\in I}\{v/U\} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\vec{\tau}$ as required.

2. $x = h_k$ for some $k \in I$. Then by Part 1 of *runtime substitution* we conclude that $\forall i \in I \backslash k, \Gamma; \Sigma_i \vdash h_i\{v/U\} : \tau_i, \Gamma; \Sigma_i \vdash h_k\{v/U\} : \tau_k$. Apply the case rule to get $\Theta; \Gamma \vdash s\,[S] : \vec{h_{ii\in I}}\{v/U\} \triangleright_{i\in I} \Sigma_i \odot s : [S]\,\vec{\tau}$ as required.

Case (Sh-Config) Let $\Theta; \Gamma \cdot x : U \vdash a\,[\langle S \rangle] : \vec{s_{ii\in I}} \triangleright_{i\in I} \Sigma_i \odot_{i\in I} s_i : [S_i]\,\varepsilon \odot a$ and $\Gamma \vdash v \triangleright U$. By case rule we conclude that $\forall i \in I, \Gamma \cdot x : U; \Sigma_i \vdash s_i : S_i$. Apply Part 1 of the *runtime substitution* to get $\Gamma; \Sigma_i \vdash s_i\{v/U\} : S_i$. Apply the case rule to get $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_{ii\in I}}\{v/x\} \triangleright_{i\in I} \Sigma_i \odot_{i\in I} s_i : [S_i]\,\varepsilon \odot a$.

Induction Hypothesis:

If $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Delta$ and $\Gamma \vdash v \triangleright U$ then $\Theta; \Gamma \vdash P\{v/x\} \triangleright \Delta$.

Induction Step:

In all cases assume that $\Gamma \vdash v \triangleright U$.
Case (Restr-sess) Let $\Theta; \Gamma \cdot x : U \vdash (\boldsymbol{\nu}\ S)\ P \triangleright \Delta$. By last rule applied we have that $\Theta; \Gamma \cdot x : U \vdash P \triangleright \Delta \cdot s : (S_1, [S_i]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2})$. Apply the induction hypothesis and then the case rule to get $\Theta; \Gamma \cdot \vdash (\boldsymbol{\nu}\ S)\ P\{v/U\} \triangleright \Delta$ as required.

Case (Restr-chan) Similar to (Restr-sess) case.

Part 3

Basic Step:

Case (Ses-Config) Let $\Theta; \Gamma \vdash s\,[S] : \vec{h_{ii\in I}} \triangleright_{i\in I} \Sigma_i \odot s : [S]\,\vec{\tau}$. By rule (Ses-Config) we conclude that $\forall i \in I, \Gamma; \Sigma_i \vdash h_i : \tau_i$. Two cases apply:

1. $x \neq h_i, \forall i \in I$. Then $\forall i \in I, \Gamma; \Sigma_i \vdash h_i\{s/x\} : \tau_i$. Apply the case rule to get $\Theta; \Gamma \vdash s\,[S] : \vec{h_{ii\in I}}\{s/x\} \triangleright_{i\in I} \Sigma_i \odot s : [S]\,\vec{\tau}$ as required.

2. $x = h_k$ for some $k \in I$. Then we can conclude that $\forall i \in I \backslash k, \Gamma; \Sigma_i \vdash h_i\{s/x\} : \tau_i, \Gamma; \Sigma_i \vdash h_k\{s/x\} : \tau_k$. Apply the case rule to get $\Theta; \Gamma \vdash s\,[S] : \vec{h_{ii\in I}}\{s/x\} \triangleright_{i\in I} \Sigma_i \odot s : [S]\,\vec{\tau}$ as required.

Case (Sh-Config) Let $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_{ii\in I}} \triangleright_{i\in I} \Sigma_i \odot_{i\in I} s_i : [S_i]\,\varepsilon \odot a$. By case rule we conclude that $\forall i \in I, \Gamma; \Sigma_i \vdash s_i : S_i$. Then $\Gamma; \Sigma_i \vdash s_i\{s/x\} : S_i$. Apply the case rule to get $\Theta; \Gamma \vdash a\,[\langle S \rangle] : \vec{s_{ii\in I}}\{s/x\} \triangleright_{i\in I} \Sigma_i \odot_{i\in I} s_i : [S_i]\,\varepsilon \odot a$.

Induction Hypothesis:

If $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : S$ then $\Theta; \Gamma \vdash P\{s/x\} \triangleright \Delta \cdot s : S$.

Induction Step:

In all cases assume that $\Gamma \vdash v \triangleright U$.

Case (Restr-sess) Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\ S)\ P \triangleright \Delta \cdot x : S$. By last rule applied we have that $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : S \cdot s : (S_1, [S_i]\,\vec{\tau_1}) \cdot \overline{s} : (S_2, [S_2]\,\vec{\tau_2})$. Apply the induction hypothesis and then the case rule to get $\Theta; \Gamma \cdot \vdash (\boldsymbol{\nu}\ S)\ P\{s/x\} \triangleright \Delta \cdot s : S$ as required.

Case (Restr-chan) Similar to (Restr-sess) case.

■

6.2.7 Subject Congruence Lemma

The *Subject Congruence* lemma states that congruence processes have the same typing.

Lemma 6.2.7 *Subject Congruence*

If $\Theta; \Gamma \vdash P \triangleright \Sigma$ and $P \equiv Q$ then $\Theta; \Gamma \vdash Q \triangleright \Sigma$

The proof considers every case in the structural congruence definition. We show an equivalance between the typing of each side of every congruence case.
Proof.

Case (α-Equivalence) Trivial proof.

Case (Idempotence) $P \equiv P \mid \mathbf{0}$. We want to show that if and only if $\Theta; \Gamma \vdash P \triangleright \Sigma$ then $\Theta; \Gamma \vdash P \mid \mathbf{0} \triangleright \Sigma$ to make the lemma statement true.

Let $\Theta; \Gamma \vdash P \mid \mathbf{0} \triangleright \Sigma \cdot \Sigma'$ with Σ completed and $\Theta; \Gamma \vdash P \triangleright \Sigma$. By applying weakening to the last judgement we have that $\Theta; \Gamma \vdash P \triangleright \Sigma \cdot \Sigma'$ to obtain the one direction of the preposition.

For the other direction consider $\Theta; \Gamma \vdash P \triangleright \Sigma$ and $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma'$ with Σ completed. By applying strengthening to the last judgement we can get $\Theta; \Gamma \vdash \mathbf{0} \triangleright \emptyset$. Finally apply the (T-Par) rule to get $\Theta; \Gamma \vdash P \mid \mathbf{0} \triangleright \Sigma$ for the other direction.

Case (Commutativity), (Assosiativity) are proven due to commutatitivity and assosiativity of the set union operator $(\cdot)$.

Case (Shared Channels)

1. $(\boldsymbol{\nu}\, a)\ \mathbf{0} \equiv \mathbf{0}$. We want to show that if and only if $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ \mathbf{0} \triangleright \Sigma$ then $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$ to make the lemma statement true.

   Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ \mathbf{0} \triangleright \Sigma$. By rule (T-Restr-a) we conclude $\Theta; \Gamma \cdot a : \langle S \rangle \vdash \mathbf{0} \triangleright \Sigma$. Since $a \notin \text{fn}(\mathbf{0})$ then we can apply *strengthening* to get $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$ as required.

   Let $\Theta; \Gamma \vdash \mathbf{0} \triangleright \Sigma$. Since $a \notin \text{fn}(\mathbf{0})$ then we can apply *weakening* to get $\Theta; \Gamma \cdot a : \langle S \rangle \vdash \mathbf{0} \triangleright \Sigma$. By rule (T-Restr-a) we conclude $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ \mathbf{0} \triangleright \Sigma$.

2. $(\boldsymbol{\nu}\, a)\ P \mid Q \equiv (\boldsymbol{\nu}\, a)\ (P \mid Q)$ if $a \notin \text{fn}(Q)$. We want to show that if and only if $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ P \mid Q \triangleright \Sigma$ then $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ (P \mid Q) \triangleright \Sigma$ to make the lemma statement true.

   Let $\Theta; \Gamma \vdash (\boldsymbol{\nu} a)\ P \mid Q \triangleright \Sigma \cdot \Sigma'$ then by (T-Par) we have $\Theta; \Gamma \vdash (\boldsymbol{\nu} a)\ P \triangleright \Sigma, \Theta; \Gamma \vdash Q \triangleright \Sigma'$. Apply (T-Restr-a) to the first judgement and *weakening* to the second judgement, since $a \notin \text{fn}(Q)$, to get $\Theta; \Gamma \cdot a : \langle S \rangle \vdash P \triangleright \Sigma$ and $\Theta; \Gamma \cdot a : \langle S \rangle \vdash Q \triangleright \Sigma'$. Apply rule (T-Par) to get $\Theta; \Gamma \cdot a : \langle S \rangle \vdash P \mid Q \triangleright \Sigma \cdot \Sigma'$. To get the required result we apply (T-Restr-a) $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ P \mid Q \triangleright \Sigma \cdot \Sigma'$.

   Let $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ (P \mid Q) \triangleright \Sigma \cdot \Sigma'$ Apply (T-Restr-a) to get $\Theta; \Gamma \cdot a : \langle S \rangle \vdash P \mid Q \triangleright \Sigma \cdot \Sigma'$, then by (T-Par) we have $\Theta; \Gamma \cdot a : \langle S \rangle \vdash P \triangleright \Sigma$ and $\Theta; \Gamma \cdot a : \langle S \rangle \vdash Q \triangleright \Sigma'$. Apply (T-Restr-a) to the first judgement and *strengthening* to the second judgement, since $a \notin \text{fn}(Q)$, to get $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma, \Theta; \Gamma \vdash Q \triangleright \Sigma'$. Apply rule (T-Par) $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ P \mid Q \triangleright \Sigma \cdot \Sigma'$ to get the required result.

3. $(\boldsymbol{\nu}\, a)\ \texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D\ \texttt{in}\ (\boldsymbol{\nu}\, a)\ P$ if $a \notin \text{fn}(D)$. We want to show that if and only if $\Theta; \Gamma \vdash (\boldsymbol{\nu}\, a)\ \texttt{def}\ D\ \texttt{in}\ P \triangleright \Sigma$ then $\Theta; \Gamma \vdash \texttt{def}\ D\ \texttt{in}\ (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma$ to make the lemma statement

true.

Let $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, a)\ \texttt{def}\ D\ \texttt{in}\ P \triangleright \Sigma$ then by rule (T-Restr-a) we have $\Theta;\Gamma{\cdot}a : \langle S\rangle \vdash \texttt{def}\ D\ \texttt{in}\ P \triangleright \Sigma$ and by rule (T-Def-in) we have $\Theta';\Gamma'{\cdot}a : \langle S\rangle \vdash D \triangleright \Sigma', \Theta';\Gamma{\cdot}a : \langle S\rangle \vdash P \triangleright \Sigma$. Apply *strengthening* for the first judgement since $a \notin \mathrm{fn}(Q)$ and rule (T-Restr-a) to the last judgement and get $\Theta';\Gamma' \vdash D \triangleright \Sigma', \Theta';\Gamma \vdash (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma$. Rule (T-Def-in) will give us $\Theta;\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma$ as required.

Let $\Theta;\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma$ then by rule (T-Def-in) we have $\Theta';\Gamma' \vdash D \triangleright \Sigma', \Theta';\Gamma \vdash (\boldsymbol{\nu}\, a)\ P \triangleright \Sigma$. Apply *weakening* to the first judgement, since $a \notin \mathrm{fn}(Q)$ and rule (T-Restr-a) to the last judgement to get $\Theta';\Gamma{\cdot}a : \langle S\rangle \vdash D \triangleright \Sigma', \Theta;\Gamma{\cdot}a : \langle S\rangle \vdash P \triangleright \Sigma$. Rule (T-Def-in) will give us $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, a)\ \texttt{def}\ D\ \texttt{in}\ P \triangleright \Sigma$.

Case (Session Channels) Same argumentation with (Shared Channels). Instead of rule (T-Restr-a) use runtime rule (Restr-sess) to obtain conclusions.

Case (Agent Definition Scopes)

1. $\texttt{def}\ D\ \texttt{in}\ \mathbf{0} \equiv \mathbf{0}$. We need to show that both sides have the same typing. By rule (T-Nill) we have that $\Theta;\Gamma \vdash \mathbf{0} \triangleright \Sigma, \Sigma$ completed. And $\forall \mathbf{X}\langle \tilde{x}\rangle = P \in D, \Theta{\cdot}\mathbf{X} : \tilde{U}\tilde{S}; \Gamma{\cdot}\tilde{x}_U : \tilde{U} \vdash P \triangleright \tilde{x}_S : \tilde{S}$. Apply rule (T-Def-in) to get $\Theta;\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ \mathbf{0} \triangleright \Sigma, \Sigma$ complete as required.

2. $(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \equiv \texttt{def}\ D\ \texttt{in}\ P \mid Q$. Following the above argumentation we have that $\Theta;\Gamma \vdash Q \triangleright \Sigma$ and if $\Theta';\Gamma \vdash P \triangleright \Sigma'$ then $\Theta;\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ P \triangleright \Sigma'$. Apply the (T-Par) rule to get $\Theta;\Gamma \vdash (\texttt{def}\ D\ \texttt{in}\ P) \mid Q \triangleright \Sigma{\cdot}\Sigma'$. For the second side of the congruence we have that if we *weaken* (6.2.1) the typing for Q we get $\Theta';\Gamma \vdash Q \triangleright \Sigma$ and by rule (T-Par) we have that $\Theta';\Gamma \vdash P \mid Q \triangleright \Sigma{\cdot}\Sigma'$. Apply rule (T-Def-in) to get $\Theta;\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ P \mid Q \triangleright \Sigma{\cdot}\Sigma'$ as required.

■

6.2.8 Environment Lemma

In this part we investigate some properties of the definitions of *composition* (4.6.2), *ordering* relation (6.1.3) and the concatenate operator $(\cdot)$ for mappings. This lemma is used for proving the Subject Reduction Theorem.

Lemma 6.2.8 *Environment Properties*

1. $\Delta \odot \Delta' = \Delta' \odot \Delta$
2. $\Delta_1 \odot (\Delta_2 \odot \Delta_3) = (\Delta_1 \odot \Delta_2) \odot \Delta_3$
3. If $\Sigma{\cdot}\Sigma'$ is defined then $\Sigma \odot \Sigma'$ is defined and $\Sigma{\cdot}\Sigma' = \Sigma \odot \Sigma'$
4. $\Delta \sqsubseteq \Delta$
5. If $\Delta \odot a$ is *well configured* then Δ is *well configured.*
6. If $\Delta \odot a \sqsubseteq \Delta' \odot a$ then $\Delta \sqsubseteq \Delta'$
7. If Δ is *well configured* and $\Delta \sqsubseteq \Delta'$ then Δ' is *well configured.*

Proof.

Part 1

By definition $\Delta \odot \Delta' = \{s : (\Delta(s), \Delta'(s)) | s \in \text{dom}(\Delta) \cap \text{dom}(\Delta')\} \cup \Delta \backslash \text{dom}(\Delta') \cup \Delta' \backslash \text{dom}(\Delta)$. By set operator properties we have that the above is equal to $\{s : (\Delta'(s), \Delta(s)) | s \in \text{dom}(\Delta') \cap \text{dom}(\Delta)\} \cup \Delta' \backslash \text{dom}(\Delta) \cup \Delta \backslash \text{dom}(\Delta') = \Delta' \odot \Delta$

Part 2

Expand the two sides of the equation with the *composition* definition (4.6.2). Apply set operator properties. The two sides should be equal.

Part 3

Let $\Sigma \cdot \Sigma'$ defined, then if $s \in \text{dom}(\Sigma)$ and $s \in \text{dom}(\Sigma')$ then $\Sigma(s) = \Sigma(s')$. This holds from the definition of the $\cdot$ operator ($s : S_1$ does not concatenate with $s : S_2$). So we can conclude that $\text{dom}(\Sigma) \cap \text{dom}(\Sigma') = \emptyset$. By the definition of the $\odot$ (4.6.2) operator we have that $\Sigma \odot \Sigma'$ is defined and $\Sigma \odot \Sigma' = \{s : (\Sigma(s), \Sigma'(s)) | s \in \text{dom}(\Sigma) \cap \text{dom}(\Sigma')\} \cup \Sigma \backslash \text{dom}(\Sigma') \cup \Sigma' \backslash \text{dom}(\Sigma) = \text{dom}(\Sigma')\} \cup \Sigma \backslash \text{dom}(\Sigma') \cup \Sigma' \backslash \text{dom}(\Sigma)$. But this is the definition of the $\cdot$ operator.

Part 4

From the definition of *ordering* (6.1.3) we have that $\emptyset \sqsubseteq \emptyset$ and $\Delta \odot \emptyset$ is defined, so $\Delta \odot \emptyset \sqsubseteq \Delta \odot \emptyset$ and by composition defintion (4.6.2)we have that $\Delta \odot \Delta$.

Part 5

Let $\Delta \odot a$ *well configured*. From the definition of *well configured* typing we have that $\forall s \in \text{dom}(\Delta \odot a), s$ is *well configured*. It is clear that $s \in \text{dom}(\Delta), \forall s \in \text{dom}(\Delta \odot a)$. So we conclude that Δ is *well configured*.

Part 6

Straightforwarg by the last premise of the definition of *ordering* (6.1.3).

Part 7

Let Δ is *well configured* and $\Delta \sqsubseteq \Delta'$. Then $\forall\, s' \in \text{dom}(\Delta), \text{wc}(\Delta, s'), \exists\, s \in \text{dom}(\Delta) \cap \text{dom}(\Delta'), \Delta(s) \sqsubseteq \Delta(s)$ and $\Delta'' \odot \Delta(s') \sqsubseteq \Delta'' \odot \Delta'(s')$ so we conclude that $\text{wc}(\Delta, s')$ and $\Delta = \Delta_1 \odot s : (S_1, [S_1'] \vec{\tau}_1) \odot \overline{s} : (S_2, [S_2'] \vec{\tau}_2)$ and $S_i' \leq S_i, S_i'' = S_i - \vec{\tau}_i, i \in \{1, 2\}, S_1'' \leq \overline{S_2''}$. From the *composition* definition (4.6.2)we have that $\Delta = \Delta_1 \odot s : S_1 \odot s : [S_1'] \vec{\tau}_1 \odot \overline{s} : S_2 \odot \overline{s} : [S_2'] \vec{\tau}_2$. So $\Delta_1 \odot s : S_1 \odot s : [S_1'] \vec{\tau}_1 \odot \overline{s} : S_2 \odot \overline{s} : [S_2'] \vec{\tau}_2 \sqsubseteq \Delta'' \odot \Delta'(s')$.By the last equation and for each of the cases of the *ordering* relation (6.1.3) we observe that wc(Δ',s') and Δ' is *well configured*.

■

6.2.9 Shared Endpoint Lemma

If a shared endpoint name appears in a process typing then there is a parallel composition with a shared endpoint structure in the process.

Lemma 6.2.9 *Shared Endpoint*

If $\Theta; \Gamma \vdash P \triangleright \Delta \odot a$ then $P = Q \mid a\,[\langle S \rangle] : \vec{s},\ \Theta; \Gamma \vdash Q \triangleright \Delta', a \notin \Delta'$.

Proof.

Proof by contradiction. Assume that $P = Q \mid R$ and $Q, R \neq a\,[\langle S\rangle] : \vec{s}$ and R is not a parallel composition. $\Theta;\Gamma \vdash Q \triangleright \Delta', a \notin \Delta'$ and $\Theta;\Gamma \vdash R \triangleright \Delta''$. Since R is not a parallel composition then by observing all non parallel typing rules beside (Sh-Config) we can say that $a \notin \Delta''$. Apply rule (Par) to get $\Theta;\Gamma \vdash Q \mid R \triangleright \Delta' \odot \Delta''$ and $a \notin \Delta' \odot \Delta''$ but this is a contradiction since $\Theta;\Gamma \vdash P \triangleright \Delta \odot a$, $\Delta' \odot \Delta'' = \Delta \odot a$ and $a \in \Delta \odot a$.

■

6.3 Subject Reduction Theorem

Theorem 6.3.1 *Subject Reduction*

If $P \rightarrow P', \Theta;\Gamma \vdash P \triangleright \Delta$ and Δ *well configured* then $\Theta;\Gamma \vdash P' \triangleright \Delta'$ and $\Delta \sqsubseteq \Delta'$

The proof considers an induction of the reduction on the structure of the derivation tree constructed by the operational semantics of the algebra.

Proof.

Basic Step:

For the basic step proof we first present the reduction rule. We type the left hand side process and by using these typings and conclusion of this part we type the right hand process to show the conclusion as required.

Case (Init-req) Let $P = \overline{u}(s).Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}}$, so $P' = (\boldsymbol{\nu}\, s)\,(Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}}{\cdot}s \mid s\,[S] : \varepsilon)$.

Type P: $\Theta;\Gamma \vdash \overline{u}(s).Q \triangleright \Sigma$ for some Σ. $\forall i \in I, \Gamma;\Sigma_i \vdash \overline{s}_i : S_i$ and by (Sh-Config) we have $\Theta;\Gamma \vdash a\,[\langle S\rangle] : \vec{s_{ii\in I}} \triangleright {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot a$. Apply the (Par) rule from runtime typing to get $\Theta;\Gamma \vdash \overline{u}(s).Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}} \triangleright \Sigma{\cdot}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot a$.

Type P': $\Theta;\Gamma \vdash \overline{u}(s).Q \triangleright \Sigma$ for some Σ, so if $\Gamma \vdash u \triangleright \langle\overline{S}\rangle$ then $\Theta;\Gamma \vdash Q \triangleright \Sigma{\cdot}s : S$. $\forall i \in I, \Gamma;\Sigma_i \vdash \overline{s}_i : S_i, \Gamma;\Sigma_0 \vdash \overline{s} : \overline{S}$ and by (Sh-Config) we have $\Theta;\Gamma \vdash a\,[\langle S\rangle] : \vec{s_{ii\in I}} s \triangleright {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot \overline{s} : \overline{S} \odot \overline{s} : [\overline{S}]\,\varepsilon \odot a$. $\Theta;\Gamma \vdash s\,[S] : \varepsilon \triangleright s : [S]\,\varepsilon$. Apply the (Par) rule to the three subprocesses to get $\Theta;\Gamma \vdash Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}}{\cdot}s \mid s\,[S] : \varepsilon \triangleright (\Sigma{\cdot}s : S) \odot {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot \overline{s} : \overline{S} \odot \overline{s} : [\overline{S}]\,\varepsilon \odot s : [S]\,\varepsilon \odot a$. By applying lemma 6.2.8 and the composition definition (4.6.2) we get $\Theta;\Gamma \vdash Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}}{\cdot}s \mid s\,[S] : \varepsilon \triangleright \Sigma{\cdot}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot \overline{s} : (\overline{S}, [\overline{S}]\,\varepsilon) \odot s : (S, [S]\,\varepsilon) \odot a$. It is obvious that $S - \varepsilon = S$ and $\overline{S} - \varepsilon = \overline{S}$ and $S \leq \overline{S}$ so we apply (Restr-sess) to get $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, s)\,(Q \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}}{\cdot}s \mid s\,[S] : \varepsilon) \triangleright \Sigma{\cdot}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot a$.

Typings for P and P' are the same and that completes the case.

Case (Init-acc) Let $P = u(x).Q \mid a\,[\langle S\rangle] : s{\cdot}\vec{s_{ii\in I}}$, so $P' = Q\{s/x\} \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}} \mid \overline{s}\,[\overline{S}] : \varepsilon$.

Type P: $\Theta;\Gamma \vdash u(x).Q \triangleright \Sigma$, for some Σ. $\forall i \in I, \Gamma;\Sigma_i \vdash \overline{s}_i : S_i, \Gamma;\Sigma_0 \vdash \overline{s} : \overline{S}$ and by (Ses-Config) we have $\Theta;\Gamma \vdash a\,[\langle S\rangle] : s{\cdot}\vec{s_{ii\in I}} \triangleright {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot \overline{s} : \overline{S} \odot \overline{s} : [\overline{S}]\,\varepsilon \odot a$. By rule (Par) we have $\Theta;\Gamma \vdash u(x).Q \mid a\,[\langle S\rangle] : s{\cdot}\vec{s_{ii\in I}} \triangleright \Sigma \odot {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot \overline{s} : \overline{S} \odot \overline{s} : [\overline{S}]\,\varepsilon \odot a$.
Type P': If $\Gamma \vdash a \triangleright \langle S\rangle$ then $\Theta;\vdash;Q \vdash \triangleright\Sigma{\cdot}x : \overline{S}$. By *substitution* lemma (6.2.5) we have that $\Theta;\vdash;Q \vdash \{\overline{s}/x\} \triangleright \Sigma{\cdot}\overline{s} : \overline{S}$. $\forall i \in I, \Gamma;\Sigma_i \vdash \overline{s}_i : S_i$ and by (Sh-Config) we have $\Theta;\Gamma \vdash a\,[\langle S\rangle] : \vec{s_{ii\in I}} \triangleright {}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot a$. By rule (Ses-Config) $\Theta;\Gamma \vdash \overline{s}\,[\overline{S}] : \varepsilon) \triangleright \overline{s}[\overline{S}]\,\varepsilon :$. Apply (Par) rule for all three subprocesses to get $\Theta;\Gamma \vdash Q\{s/x\} \mid a\,[\langle S\rangle] : \vec{s_{ii\in I}} \mid \overline{s}\,[\overline{S}] : \varepsilon \triangleright (\Sigma{\cdot}\overline{s} : \overline{S}) \odot ({}_{i\in I}\Sigma_i \odot_{i\in I} \overline{s_i} : [S_i]\,\varepsilon \odot a) \odot \overline{s} : [\overline{S}]\,\varepsilon$.

Apply lemma 6.2.8 to verify that the two typings for P and P' are the equal as required by the theorem.

Case (R-Send) $P = s\,!\langle v\rangle; Q \mid s\,[!\langle T\rangle; S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'}$, so $P' = Q \mid s\,[S] : \vec{h}_{ii\in I} \mid \overline{s}\,[S'] : \vec{h'}_{ii\in I'} v$. Two cases apply:

1. $T = U$. By typing P we have that if $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S$ then by (T-Send-u) we have that $\Gamma \vdash v \triangleright U, \Theta;\Gamma \vdash s\,!\langle v\rangle; Q \triangleright \Sigma\cdot s :!\langle U\rangle; S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[!\langle U\rangle; S] : \vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [!\langle U\rangle; S]\,\tau_i$. If $\forall i \in I', \Gamma;\Sigma'_i \vdash h'_i : \tau'_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash \overline{s}\,[S'] : \vec{h'}_{ii\in I'} \triangleright {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i$. Apply (Par) and get $\Theta;\Gamma \vdash s\,!\langle v\rangle; Q \mid s\,[!\langle U\rangle; S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'} \triangleright (\Sigma\cdot s :!\langle U\rangle; S) \odot ({}_{i\in I}\Sigma_i \odot s : [!\langle U\rangle; S]\,\tau_i) \odot {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i$.

   By typing P' we have that $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[S] : \vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i$. If $\forall i \in I', \Gamma;\Sigma'_i \vdash h'_i : \tau'_i, \Gamma;\Sigma'_0 \vdash v : U$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash \overline{s}\,[S'] : \vec{h'}_{ii\in I'} v \triangleright {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i U$. Apply (Par) and get $\Theta;\Gamma \vdash Q \mid s\,[S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'} v \triangleright (\Sigma\cdot s : S) \odot ({}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i) \odot {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i U$.

   If we apply lemma 6.2.8 then by definition of the *ordering* relation (6.1.3) we have that the typings of P and P' are ordered as required.

2. $T = S'$ By typing P we have that if $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S$ then by (T-Send-s) we have that $\Theta;\Gamma \vdash s\,!\langle k\rangle; Q \triangleright \Sigma\cdot s :!\langle S'\rangle; S\cdot k : S'$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[!\langle S\rangle; S] : \vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [!\langle T\rangle; S]\,\tau_i$. If $\forall i \in I', \Gamma;\Sigma'_i \vdash h'_i : \tau'_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash \overline{s}\,[S'] : \vec{h'}_{ii\in I'} \triangleright {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i$. Apply (Par) and get $\Theta;\Gamma \vdash s\,!\langle k\rangle; Q \mid s\,[!\langle S'\rangle; S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'} \triangleright (\Sigma\cdot s :!\langle S'\rangle; S\cdot k : S') \odot ({}_{i\in I}\Sigma_i \odot s : [!\langle S'\rangle; S]\,\tau_i) \odot {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i$.

   By typing P' we have that $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[S] : \vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i$. If $\forall i \in I', \Gamma;\Sigma'_i \vdash h'_i : \tau'_i, \Gamma; k : S' \vdash k : S'$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash \overline{s}\,[S'] : \vec{h'}_{ii\in I'} k \triangleright ({}_{i\in I}\Sigma'_i\cdot k : S') \odot \overline{s} : [S']\,\tau'_i S'$. Apply (Par) and get $\Theta;\Gamma \vdash Q \mid s\,[S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'} S' \triangleright (\Sigma\cdot s : S) \odot (({}_{i\in I}\Sigma_i\cdot k : S) \odot s : [S]\,\tau_i) \odot {}_{i\in I}\Sigma'_i \odot \overline{s} : [S']\,\tau'_i U$.

   If we apply lemma 6.2.8 then by definition of the *ordering* relation (6.1.3) we have that the typings of P and P' are ordered as required.

Case (R-Receive) $P = s\,?(x); Q \mid s\,[?(T); S] : v\vec{h}_{ii\in I}$, so $P' = Q\{v/x\} \mid s\,[S] : \vec{h}_{ii\in I}$. Two cases apply:

1. $T = U$. By typing P we have that if $\Theta;\Gamma\cdot x : U \vdash Q \triangleright \Sigma\cdot s : S$ then by (T-Receive-u) we have that $\Theta;\Gamma \vdash s\,?(x); Q \triangleright \Sigma\cdot s :?(U); S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i, \Gamma;\Sigma_0 \vdash v \triangleright U$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[!\langle U\rangle; S] : v\vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [!\langle U\rangle; S]\,U\tau_i$. Apply (Par) and get $\Theta;\Gamma \vdash s\,?(x); Q \mid s\,[?(U); S] : v\vec{h}_{ii\in I} \triangleright (\Sigma\cdot s :?(U); S) \odot ({}_{i\in I}\Sigma_i \odot s : [?(U); S]\,U\tau_i)$.

   By typing P' we have that $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S$. By *substitution* lemma (6.2.5) we have that $\Theta;\Gamma \vdash Qvx \triangleright \Sigma\cdot s : S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[S] : \vec{h}_{ii\in I} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i$. Apply (Par) and get $\Theta;\Gamma \vdash Q\{v/x\} \mid s\,[S] : \vec{h}_{ii\in I} \mid s\,[S'] : \vec{h'}_{ii\in I'} v \triangleright (\Sigma\cdot s : S) \odot ({}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i)$.

   If we apply lemma 6.2.8 then by definition of the *ordering* relation (6.1.3) we have that the typings of P and P' are ordered as required.

2. $T = S'$ By typing P we have that if $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S\cdot x : S'$ then by (T-Receive-s) we have that $\Theta;\Gamma \vdash s\,?(k);Q \triangleright \Sigma\cdot s :?(S');S$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i, \Gamma; v : S' \vdash v : S'$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[?(S');S] : v\vec{h_{ii\in I}} \triangleright {}_{i\in I}\Sigma_i \odot s : [!\langle T\rangle;S]\,U\tau_i \odot v : S'$. Apply (Par) and get $\Theta;\Gamma \vdash s\,?(x);Q \mid s\,[!\langle S'\rangle;S] : v\vec{h_{ii\in I}} \triangleright (\Sigma\cdot s :?(S');S) \odot ({}_{i\in I}\Sigma_i \odot s : [?(S');S]\,U\tau_i)\cdot v : S'$.

   By typing P' we have that $\Theta;\Gamma \vdash Q \triangleright \Sigma\cdot s : S\cdot x : S'$. By *substitution* lemma (6.2.5) wwe have that $\Theta;\Gamma \vdash Q\{v/x\} \triangleright \Sigma\cdot s : S\cdot v : S'$. If $\forall i \in I, \Gamma;\Sigma_i \vdash h_i : \tau_i$ then by rule (Ses-Config) we have $\Theta;\Gamma \vdash s\,[S] : \vec{h_{ii\in I}} \triangleright {}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i$. Apply (Par) and get $\Theta;\Gamma \vdash Q\{v/x\} \mid s\,[S] : \vec{h_{ii\in I}} \triangleright (\Sigma\cdot s : S\cdot v : S') \odot ({}_{i\in I}\Sigma_i \odot s : [S]\,\tau_i)$.

   If we apply lemma 6.2.8 then by definition of the *ordering* relation (6.1.3) we have that the typings of P and P' are ordered as required.

Case (R-Select) Similar to the first case of case (R-Send).

Case (R-Branch) Similar to the first case of case (R-Receive).

Case (R-If-true), (R-If-false) are straightforward. By (T-If) rule we conclude that boht sides of the reduction have the same runtime typing as required.

Case (R-Typecase) Let $P = \texttt{typecase}\ s\ \texttt{of}\ \{(X_i)S_i : P_i\}_{i\in I}$, so $P' = P_i$.

Type P by (T-Typecase) and get $\Theta;\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{(X_i)S_i : P_i\}_{i\in I} \triangleright \Sigma\cdot s : [S_i]_{i\in I}$. For some Σ. If we use (T-Typecase) on P we conlcude that $\Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot s : S_i$. It is obvious that $[S_i]_{i\in I} \leq S_i$, so $\Sigma\cdot s : [S_i]_{i\in I} \leq \Sigma\cdot s : S_i$. By applying the (T-Sub) rule we have that $\Theta;\Gamma \vdash P_i \triangleright \Sigma\cdot s : [S_i]_{i\in I}$ as required.

Induction Hypothesis:

If $P \to P', \Theta;\Gamma \vdash P \triangleright \Delta$ and Δ *well configured* then $\Theta;\Gamma \vdash P' \triangleright \Delta'$ and $\Delta \sqsubseteq \Delta'$.

Type P: $\Theta;\Gamma' \vdash Q \triangleright \Delta$ for some Δ and by (T-Process-var) we have that $\Theta';\Gamma \vdash \mathbf{X}\langle\tilde{v}\rangle \triangleright \tilde{v}_s : \tilde{S}$. Apply (Par) rule and get $\Theta';\Gamma \vdash \mathbf{X}\langle\tilde{v}\rangle | Q \triangleright \tilde{v}_s : \tilde{S} \odot \Delta$. With (T-Def-in) we get $\Theta;\Gamma \vdash \texttt{def}\ D\texttt{in}\ (\mathbf{X}\langle\tilde{v}\rangle \mid Q) \triangleright \tilde{v}_s : \tilde{S} \odot \Delta$.
Type P': $\tilde{x} = R \in D$. If we substitute the process variable $\mathbf{X}$ with R we have that $\Theta';\Gamma \vdash R\langle\tilde{x}\rangle \triangleright \tilde{x}_s : \tilde{S}$ and by *substitution* (6.2.5) we have that $\Theta';\Gamma \vdash R\langle\tilde{x}\rangle\{\tilde{v}/\tilde{x}\} \triangleright \tilde{v}_s : \tilde{S}$. Apply the (T-Def-in) to get $\Theta;\Gamma \vdash \texttt{def}\ D\texttt{in}\ (\mathbf{P}\langle\tilde{v}\rangle \mid Q) \triangleright \tilde{v}_s : \tilde{S} \odot \Delta$ as required.
Induction Step:

Case (R-Par) Let $P = R \mid Q$ and $P' = R' \mid Q$. From rule (R-Par) conclude that $R \to R'$. By the induction hypothesis we have that if Δ is *well configured* and $\Theta;\Gamma \vdash R \triangleright \Delta$ then $\Theta;\Gamma \vdash R' \triangleright \Delta'$ and $\Delta \sqsubseteq \Delta'$.

Type P: If $\Theta;\Gamma \vdash Q \triangleright \Delta_Q$ then by rule (Par) we have that $\Theta;\Gamma \vdash R \mid Q \triangleright \Delta \odot \Delta_Q$.

Type P': By rule (Par) $\Theta;\Gamma \vdash R' \mid Q \triangleright \Delta' \odot \Delta_Q$.

Straightforward by the definition of *ordering* (6.1.3) we have that $\Delta \odot \Delta_Q \sqsubseteq \Delta' \odot \Delta_Q$ because $\Delta \sqsubseteq \Delta'$.

Case (R-Restr-a) Let $P = (\boldsymbol{\nu}\ a)\ Q$ and $P' = (\boldsymbol{\nu}\ a)\ Q'$. $\Theta;\Gamma \vdash (\boldsymbol{\nu}\ a)\ Q \triangleright \Delta$, for some Δ. Apply the case rule to get $\Theta;\Gamma \vdash Q \triangleright \Delta \odot a$. Apply lemma 6.2.9 to get $\Theta;\Gamma \vdash Q' \mid a\,[\langle s\rangle] : \vec{s} \triangleright \Delta \odot a$. No action can create or destroy a shared endpoint so by the induction hypothesis we have that $\Theta;\Gamma \vdash Q'' \mid a\,[\langle s\rangle] : \vec{s} \triangleright \Delta' \odot a$ and $\Delta \odot a \sqsubseteq \Delta' \odot a$. By lemma 6.2.8 we have that $\Delta \sqsubseteq \Delta'$. Apply

the case rule to get $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, a)\, Q' \triangleright \Delta'$ as required.

Case (R-Restr-s) Let $P = (\boldsymbol{\nu}\, s)\, Q$ and $P' = (\boldsymbol{\nu}\, s)\, Q'$. $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, s)\, Q \triangleright \Delta$, for some Δ. Apply the case rule to get $\Theta;\Gamma \vdash Q \triangleright \Delta \cdot s : [S_1]\, S_1\vec{\tau_1} \cdot \overline{s} : [S_2]\, S_2\vec{\tau_2}$. Before we apply the induction hypothesis we must consider the possible transitions:

1. Transitions the s endpoint: Apply induction hypothesis to get $\Theta;\Gamma \vdash Q' \triangleright \Delta \cdot s : [S_1']\, S_1'\vec{\tau_1'} \cdot \overline{s} : [S_2']\, S_2'\vec{\tau_2'}$. Apply the case rule to get $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, s)\, Q' \triangleright \Delta'$ as required.

2. Any other transition: Apply induction hypothesis to get $\Theta;\Gamma \vdash Q' \triangleright \Delta' \cdot s : [S_1]\, S_1\vec{\tau_1} \cdot \overline{s} : [S_2]\, S_2\vec{\tau_2}$ with $\Delta \cdot s : [S_1]\, S_1\vec{\tau_1} \cdot \overline{s} : [S_2]\, S_2\vec{\tau_2} \sqsubseteq \Delta' \cdot s : [S_1]\, S_1\vec{\tau_1} \cdot \overline{s} : [S_2]\, S_2\vec{\tau_2}$. By the *ordering* definition (6.1.3) we conclude that $\Delta \sqsubseteq \Delta'$. Apply the case rule to get $\Theta;\Gamma \vdash (\boldsymbol{\nu}\, s)\, Q' \triangleright \Delta'$ as required.

Case (R-Instance) Let $P = \texttt{def}\ D \texttt{in}\ (\mathbf{X}\langle \tilde{v} \rangle \mid Q)$ and $P' = \texttt{def}\ D \texttt{in}\ (R\{\tilde{v}/\tilde{x}\} \mid Q)$. By the case rule we have that $\mathbf{X}\langle \tilde{x} \rangle = R \in D$.

Case (R-Def-scope) Straightforward by application of the case rule, followed by the induction hypothesis and then the case rule to get the desired result.

Case (R-Struct) Straightforward by application (R-Struct), followed by the induction hypothesis and the application of the *subject congruence* (6.2.7) lemma both sides typings.

■

Corollary 6.3.2 If $P \rightarrow^* P', \Theta;\Gamma \vdash P \triangleright \Delta$ and Δ *well configured* then $\Theta;\Gamma \vdash P' \triangleright \Delta'$ and Δ' and Δ' is *well configured.*

The above is a corollary of the Subject Reduction theorem. It shows that given any transition sequence on a process with *well configured* typing results in a process with *well configured* typing. The proof is a simple induction on the transition length.

Proof.

Basic Step:

For no transitions it is trivial to show that the result holds.

Induction Hypothesis:

If $P \rightarrow^* P', \Theta;\Gamma \vdash P \triangleright \Delta$ and Δ *well configured* then $\Theta;\Gamma \vdash P' \triangleright \Delta'$ and Δ' is *well configured.*

Induction Step:

Let $P \rightarrow^* P'$ and $P' \rightarrow P''$. From the induction hypothesis we have that $\Theta;\Gamma \vdash P' \triangleright \Delta$ and Δ is *well configured.* From Subject Reduction we have that $\Theta;\Gamma \vdash P'' \triangleright \Delta'$ and $\Delta \sqsubseteq \Delta'$. From lemma 6.2.8 Δ' is *well configured* as required.

■

6.4 Type Safety Theorem

The second important property we need to investigate is that of *Type Safety.* This theorem says that a *well configured* typable program never reduces to an error.

6.4.1 Error Process

Before we proceed we define some notions about process prefixes:

Definition 6.4.1 *Process prefixes*

1. $p \quad ::= \quad u(x). \mid \overline{u}(s). \mid ps$
2. $ps \quad ::= \quad pio \mid psb$
3. $pio \quad ::= \quad s\,!\langle v\rangle; \mid s\,?(x);$
4. $psb \quad ::= \quad s \triangleleft l_k; \mid s \triangleright \{l_i : P_i\}$
5. $pt \quad ::= \quad !\langle T\rangle; \mid ?(T); \mid \oplus\{l_i : S_i\}_{i\in I} \mid \&\{l_i : S_i\}_{i\in I}$
6. A session prefix has *correspondence* to session type prefix as follows*:*
   - $s\,!\langle v\rangle;$ *corresponds to* $!\langle T\rangle;$.
   - $s\,?(x);$ *corresponds to* $?(T);$.
   - $s \triangleleft l_i; P$ *corresponds to* $\oplus\{l_i : S_i\}_{i\in I}$.
   - $s \triangleright \{l_i : P_i\}$ *corresponds to* $\&\{l_i : S_i\}_{i\in I}$.

■

We define the processes that are considered as errors. A well configured process should not reduce to a process of this form. The typing system we have specified should be able to caught processes that reduce to these errors.

Definition 6.4.2 *Error Process*

If $(\boldsymbol{\nu}\ \vec{u})\ (\boldsymbol{\nu}\ \vec{k})\ P = (\boldsymbol{\nu}\ \vec{u})\ (\boldsymbol{\nu}\ \vec{k})\ Q \mid R$ then P *is an error* whenever Q is equal to*:*

1. $pio; P' \mid s\,[pt; S] : \vec{h}, pio$ *does not correspond to* pt.
2. $psb \mid s\,[pt] : \vec{h}, psb$ *does not correspond to* pt.
3. $s\,!\langle v\rangle; P' \mid s\,[!\langle T\rangle; S] : \vec{h}, \Gamma \vdash v \triangleright T', T \not\leq T'$.
4. $s\,?(x); P' \mid s\,[?(T); S] : \vec{h}, \Gamma \vdash x \triangleright T', T' \not\leq T$.
5. $s\,?(x); P' \mid s\,[?(T); S'] : v\vec{h}, \Gamma \vdash x \triangleright T', \Gamma \vdash v \triangleright T, T' \not\leq T$.
6. $s \triangleleft l_k; P' \mid s\,[\oplus\{l_i : S_i\}_{i\in I}] : \vec{h},\ l_k \notin \{l_i\}_{i\in I}$.
7. $s \triangleright \{l_i : P_i\}_{i\in I} \mid s\,[\&\{l_i : S_i\}_{i\in I'}] : \vec{h}, \{l_i\}_{i\in I} \not\subseteq \{l_i\}_{i\in I'}$.
8. $s \triangleright \{l_i : P_i\}_{i\in I} \mid s\,[S'] : v\vec{h}, \Gamma \vdash v \triangleright T, T \neq \mathsf{lab}$.
9. $ps; P' \mid s\,[end] : \vec{h}$.
10. $p; s\,[S] : \vec{h}$.
11. $p; a\,[\langle S\rangle] : \vec{s}$.

■

6.4.2 Type Safety Theorem

Theorem 6.4.3 *Type Safety*

If P is a program and $\Theta;\Gamma \vdash P \triangleright \Delta$ with Δ *well configured* and $P \rightarrow^* P'$ then P' is not *an error process.*

Proof.

The proof is done by using contradiction and the Subject Reduction theorem (6.3.1). The general pattern of the proof will follow. Then we will analyze each *error process* case in order to find the properties that make the general proof true.

Assume that $Q \rightarrow^* P'$ and P' is an *error process.* An *error process* is untypable or $\Theta;\Gamma \vdash P' \triangleright \Delta, \Delta$ not *well configured.* By the corollary of Subject Reduction (6.3.1) we have that if $\Theta;\Gamma \vdash P' \triangleright \Delta$ for some *well configured* Δ. But this is a contradiction because P' is untypable or Δ is *well configured* so the initial assumption that P' is an *error process* is wrong.

It remains to show that each *error process* untypable, or typable with a non *well configured* typing.

Case (Error Process 1) If $\Theta;\Gamma \vdash P' \triangleright \Sigma\cdot s : S$ then $\Theta;\Gamma \vdash pio; P' \triangleright \Sigma\cdot s : pt; S$ where *pio corresponds* to pt. Also $\Theta;\Gamma \vdash s\,[pt'; S'] : \vec{h} \triangleright_{i\in I}\Sigma_i \odot s : (pt'; S', [\vec{\tau}\,])$. Note that $pt \neq pt'$ cause of the restriction that *pio* does not *correspond* to pt'. If we apply the (Par) we must use the *composition* relation, that states that $pt; S \leq pt'; S'$. But this does not hold due to the subtyping relation 4.5. So the whole *error process* P is untypable as required.

Case (Error Process 2), (Error Process 9). Same argumentation as (Error Process 1).

Case (Error Process 3) If $\Theta;\Gamma \vdash P' \triangleright \Sigma\cdot s : S$ then $\Theta;\Gamma \vdash s\,!\langle v\rangle; P' \triangleright \Sigma\cdot s :!\langle T\rangle; S$. Also $\Theta;\Gamma \vdash s\,[!\langle T'\rangle; S'] : \vec{h} \triangleright_{i\in I}\Sigma_i \odot s : (!\langle T'\rangle; S', [\vec{\tau}\,])$. If we apply the (Par) we must use the *composition* relation, that states that $!\langle T\rangle; S \leq !\langle T'\rangle; S'$. But this does not hold due to the subtyping relation 4.5. So whole *error process* P is untypable as required.

Case (Error Process 4), (Error Process 6), (Error Process 7) are similar to (Error Process 2).

Case (Error Process 5) If $\Theta;\Gamma\cdot x : T \vdash P' \triangleright \Sigma\cdot s : S$ then $\Theta;\Gamma \vdash s\,?(x); P' \triangleright \Sigma\cdot s :?(T); S$. Also $\Theta;\Gamma \vdash s\,[?(T); S'] : v\vec{h} \triangleright_{i\in I}\Sigma_i\cdot\Sigma_0 \odot s : (?(T); S', [T'\vec{\tau}\,])$ and $\Gamma \vdash x \triangleright T, \Gamma \vdash v \triangleright T', T' \not\leq T$. By using the (Par) rule we have that $\Theta;\Gamma \vdash s\,?(x); P' \mid s\,[?(T); S'] : v\vec{h} \triangleright \Sigma s :?(T); S \triangleright_{i\in I}\Sigma_i\cdot\Sigma_0\cdot s : (?(T); S, [?(T); S]\,T'\vec{\tau})$. *Error process* P has the form $R \mid s\,?(x); P' \mid s\,[?(T); S'] : v\vec{h}$. If $\Theta;\Gamma \vdash R \triangleright \Delta$ then by (Par) we have that $\Theta;\Gamma \vdash R \mid s\,?(x); P' \mid s\,[?(T); S'] : v\vec{h} \triangleright \Delta \odot (\Sigma s :?(T); S \triangleright_{i\in I}\Sigma_i\cdot\Sigma_0\cdot s : (?(T); S, [?(T); S]\,T'\vec{\tau}))$. Two cases occur. $\overline{s} : (S', [S']\,t\vec{a}u') \notin \Delta, \overline{s} : (S', [S']\,t\vec{a}u') \in \Delta$. In the first case we do not have the *well configure* property. In the second case in order to have *well configure* it should hold that $S'' = ?(S); -T'\vec{\tau}$ but this is undefined since $T \neq T'$ and the *well configure* property does not hold.

Case (Error Process 8) Similar to (Error Process 5)

Case (Error Process 10), (Error Process 11). These processes cannot be typed because there is no typing rule that consider that kind of processes. *Error process* P is untypable as required.

■

6.5 Deadlock Situations

In the revised session type system [25] there are some processes that are typable but can lead to deadlock processes. These processes are not caught by the typing system and the system soundness

property does not consider them. This discussion is very important to our system as we shall see later. In order to be more concrete consider the process.

$$\overline{a}(s).\overline{b}(s').s\,!\langle s'\rangle;\mathbf{0} \mid a(x).b(y).y\,?(z);z\,!\langle 1\rangle;z\,?(num);\mathbf{0}$$

The above process is typable and balanced under the second system presented in [25]. But if we try to reduce it we end up with the process:

$$z\,!\langle 1\rangle;z\,?(num);\mathbf{0}$$

This process cannot be reduced and gets stuck without terminating. In order to catch these kinds of situations a more advance typing system should be developed. But the question we need to answer at this point is the situation where deadlock occurs.

The answer is simple. Typing system should consider that delegation changes the structure of the communication. In this system when we have session delegation we can have deadlock under conditions. In most cases after delegation a deadlock will not occur but under conditions similar to the above we can have deadlock as shown.

This deadlock situation should be happening in the system of this project. And indeed we can think of a similar process that can reach to deadlock after delegation. But this system has also a session creation mechanism that can lead to deadlock without the presence of delegation. As an example consider the process:

$$\overline{a}(s).a(x).x\,?(y);s\,!\langle 1\rangle;\mathbf{0}$$

This process should reduce to:

$$(\boldsymbol{\nu}\ s)\ (\overline{s}\,?(y);s\,!\langle 1\rangle;\mathbf{0} \mid s\,[S]:\varepsilon \mid \overline{s}\,[S]:\varepsilon)$$

But this process is a deadlock. This situation is not defined in the error processes and not caught by the typing system. The fact is that what happens here is similar to what happens with delegation in the previous systems. With delegation we have a session endpoint to be binded to a process that also has the corresponding endpoint binded. So we have the two endpoints of a session binded in the same process. This can lead to deadlocks.

The same situation happens in the event system. With the session creation mechanism we can end up with a process that has the two endpoints of a session in the bound sessions. Fortunately there is a solution for this problem. This solution is used in the examples in chapter 5.

If the above scenario occurs then a typed sending process cannot never stuck because of the asynchrony and the well configuration of the typing. When receiving a message from an empty queue and the corresponding sending is to be done by a derivative of the receiving process then we have a deadlock.

If we guard every input prefix on a session that has its endpoints binded to only one process, with an `if arrived` s `then` $s\,?(x);P$ `then` Q prefix then we can avoid deadlock. If the queue is empty then we do not read from the queue but we proceed with doing something else. If we do this for all input prefixes on all processes that have the two endpoints of a session in their bound sessions then the process cannot get stuck. The `arrived` s construct is a very powerful construct that can be use not only to handle event processing but to avoid deadlock situations. We explain this argument formally:

6.5.1 Deadlock Avoidance - Definitions and Proof

Here we define some important processes that are involved in deadlock creation.

A *binded endpoint receiver* is a process on the session input prefix, and it is the only process that has both endpoints of a session restricted to it.

Definition 6.5.1 *Binded Endpoint Receiver*

Process $(\boldsymbol{\nu}\ s, \overline{s})\ (k\,?(x); P \mid k\,[S_1] : \vec{h}_1 \mid \overline{k}\,[S_2] : \vec{h}_2)$ is called a *binded endpoint receiver.*

■

A *guarded receiver* is an *binded enpoint receiver* that has an `arrived` prefix.

Definition 6.5.2 *Guarded Receiver*

Process $(\boldsymbol{\nu}\ s, \overline{s})\ (\texttt{if arrived}\ k\ \texttt{then}\ k\,?(x); P\ \texttt{then}\ Q \mid k\,[S_1] : \vec{h}_1 \mid \overline{k}\,[S_2] : \vec{h}_2), Q \neq k\,?(x); Q'$ is called a *guarded receiver.*

■

The definition below gives the conditions where a deadlock can occur. The two endpoints are binded to one process, the process is prefixed by a session input and the receiving queue is empty. Notice that *deadlock process* is a parallel composition of an *endpoint binded receiver.*

Definition 6.5.3 *Deadlock Process*

Process $(\boldsymbol{\nu}\ s, \overline{s})\ (k\,?(x); R \mid k\,[S_1] : \varepsilon \mid \overline{k}\,[S_2] : \vec{h}_2)$ is called a *deadlock process.*

Note that a deadlock process cannot present any action:

$$(\boldsymbol{\nu}\ s, \overline{s})\ (k\,?(x); R \mid k\,[S_1] : \varepsilon \mid \overline{k}\,[S_2] : \vec{h}_2) \not\rightarrow$$

■

We now define the well guarded process.

Definition 6.5.4 *Well Guarded Process*

P is a *well guarded process* if and only if $P \rightarrow^* P' \mid Q$ and if P'' is a *binded endpoint receiver* then $\exists\ P'' \mid Q \cdot P'' \mid Q \rightarrow P' \mid Q$ and P'' is a *guarded receiver.*

■

6.5.2 Guarded Reduction

In this part we show when a process cannot reach a deadlock state.

Lemma 6.5.5 *Guarded Reduction*

If P is a *guarded receiver* and $P \rightarrow P'$ then P' is not a *deadlock process.*

Proof.

$P = (\boldsymbol{\nu}\ s, \overline{s})\ (\texttt{if arrived}\ k\ \texttt{then}\ k\,?(x); P\ \texttt{then}\ Q \mid k\,[S_1] : \vec{h}_1 \mid \overline{k}\,[S_2] : \vec{h}_2)$. Two reduction can occur: $P' = k\,?(x); P \mid k\,[S_1] : \vec{h}_1) \mid \overline{k}\,[S_2] : \vec{h}_2$ and $\vec{h_1} = v\vec{h'}_1$ by (R-Arrived-yes). The second reduction considers $P' = Q \mid k\,[S_1] : \vec{h}_1) \mid \overline{k}\,[S_2] : \vec{h}_2$. By definition of *deadlock process* and by definition of *well guarder process* we have that P' is not a *deadlock process.*

■

A *well guarded* receiver cannot ever reduce to a *deadlock process.*

Theorem 6.5.6 *Well Guarded Reduction*

If P is *well guarded* and $P \rightarrow^* P' \mid R$ then P' is not a *deadlock process.*

Proof.

We will prove the theorem by contradiction.
Assume that P is *well guarded* and $P \rightarrow^* P' \mid Q$. Furthermore P' is a *deadlock process.* Note that a *deadlock process* is an *endpoint binded receiver.* By *well guarded* definition $\exists\, P'' \mid Q \cdot P'' \mid Q \rightarrow P' \mid Q$ and P'' is a *guarded receiver.* Apply the (R-Par) to get $P'' \rightarrow P'$. By lemma 6.5.5 we have that a *guarded receiver* cannot reduce to a *deadlock process.* This is a contradiction.

■

Chapter 7

Conclusions

7.1 Conclusions

This project has developed a Session Type discipline able to handle event driven models. We have seen a study of event driven model to find the key properties that can help us introduce a theoretical model. The design of the session types system was base on an extension of previous session type systems. The extensions and some modifications from other session types system served the purpose of handling events. Some example scenarios and their typing are programmed and typed using the event driven session type calculus. A very interesting part is the part where we prove soundness and other theoretical properties for the calculus showing the consistency of the formalism.

In this part we will discuss the conclusions from the entire project. We will start from conclusions from the event discussion, followed by the designing choices for the session calculus. Examples stress out some important observations and finally the study around the calculus can give us many points we can discuss.

7.1.1 Event Driven Model

- Events driven programming is a model used to naturally describe asynchronous concurrent systems. An asynchronous message between parallel computations it is considered an event.

- The handling of events can happen in two ways. A computation spawns a new thread to process the event or a context is created for each event and the receiving applications schedules the event contexts for processing inside an instruction stream. Hybrid systems are also present. The interesting fact here is that an application uses concurrency to handle concurrent events in both approaches. In the first approach new parallel entities are created. And in the second approach new independent contexts are created. The execution of these contexts can be described by concurrent programming primitives.

- When observing the two event solutions we conclude that the thread spawning approach is actually an instance of the one instruction stream approach. The application uses the scheduling and context switching mechanisms of the operating system to handle concurrency and essentially use the one instruction stream on a lower level. Here multicore systems present some interest because we can actually have parallel execution of threads.

- Staged Event Driven Architecture is a promising architecture for building application based on events. It considers an application as pipeline stages of the different components that communicate through events. It was developed having in mind internet applications but it can also be a general architecture for building applications. As for the internet properties, it targets *well condition* systems. Well condition systems are characterized by linear increase of the throughput and response time as the parallel load increases. Also the response time should be near the average for all requests. Implementations on SEDA have left promising conclusions and we would like to be able to model SEDA applications in the theoretic framework of this calculus.

7.1.2 Design choices for an Event Driven Session Type Discipline

- For the purposes of this project we chose to implement and define in a theoretic level the single stream approach for handling events. This choice is supported by the fact that the two solutions for event driven programming can be described by this approach.

- The algebra should have constructs that can easily implement scheduling and context switching. Also we need a mechanism to help us recognize the type of an event because an application should be in place to handle different multiple events.

- The most important design decision and very critical for the development of the algebra was the way we chose to model events. Events are the central entity in event driven models. The central entity in session types systems in a session channel. It is only natural to consider events as sessions and vice versa. Session describe an interaction and in extension, since sessions are a part of π calculus, computation. The same holds for events. Each event triggers a computation. So there is no other choice than to unify these two notions in our system.

- The second important design decision has to do with asynchrony. Asynchrony defines event driven models. For this purpose we use session endpoints. Session endpoints are session type constructs, described in session type literature, that were design to give an asynchronous character to a calculus. Basic sending and receiving is done through these endpoints and they are ideal for coping with event asynchrony.

- A major concern for the model we chose to implement is that of context switching and scheduling. The idea is that an event is a series of interactions described by a session. This interaction can create a context of binded variables used in the computation. When processing an event and wait for a message to be received the event driven model proposes that the context should be stored and we must proceed with processing of another event context. This implies using data structures for storing and retrieving a context. Data structures such as caches, queues and hashes can be described in π calculus. This could make the definition more complex, disallowing us to have a simple calculus. Fortunately there is the session endpoint queue that it is a queue construct well described in the literature. A process can use a session endpoint to store and retrieve data. The problem presented here is that a process can send to an endpoint and another process can receive from an endpoint. We want a process to be able to send and receive from the same endpoint. The problem is solved by introducing a mechanism for asynchronous session creation. As described in the reduction semantics, this mechanism can achieve desired goal. For this purpose we introduce a structure with similar properties to the session endpoint configuration called shared endpoint configuration.

- Scheduling is handled by introducing the `arrived` construct. This is a simple expression used to check a session endpoint. It evaluates to true when an endpoint is non - empty, false otherwise. Using this expression in an `if arrived then ... else ...` construct can give the power to the system to decide the next action.

- Events are essentially the same as session types. If we consider that a process can be able to handle different multiple events then we want session to be able to describe different interactions. This approach is a novelty of this system, since it is the first time we want to observe a session describing many different interactions. The interaction of a session is given through its type. We need to define a type that holds many session types at the same time. The answer comes through the session set type construct. This is nothing else than a set of session types used to describe many session types at the same instance.

- Set session types naturally define a subtyping relation. Intuitively a larger set can be used in the place of a smaller set so we observe that supersets are subtypes of subsets. This subtyping is used when typing programs so it must be defined. For this purpose we defined a subtyping relation that takes into account subtyping between primitive types, subtyping between session types and subtyping between session set types.

- To recognize the type of an event through a session channel typed with a session set type we need the notion of matching. Matching is used to match the interaction carried by an event to a type described by the program. A matching tree is constructed that runs along the session type trees of the event and the program construct that can figure out if the two sessions match.

- The construct that allows the programmer to define events and recognize them through matching is the `typecase` k `of` $\{S_i : P_i\}_{i \in I}$ construct. Session k is a set session type channel. The reduction on this construct matches the event carried by k to sessions $S_i, i \in I$. The first match defines the derivative process $P_i, i \in I$. What we have here is a refinement of the general session type to a more specific session type that is processed by the derivative process $P_i, i \in I$.

7.1.3 Session Type Definition

- Definition of terminals, terms, calculus syntax and operational semantics are used for the runtime specification of the calculus. Because construct `typecase` k `of` $\{S_i : P_i\}_{i \in I}$ requires runtime matching of session types, reduction semantics take into account the session type syntax.

- Session type terms, terminals and syntax and the definition of set types and their operations complete the session type syntax. The subtyping relation considers primitive subtyping, session type subtyping and set session type subtyping. Finally the matching relation is necessary for the `typecase` k `of` $\{S_i : P_i\}_{i \in I}$ reduction.

- Session type syntax guides a typing system to build the session typing for a calculus program. The typing system is a tree that runs along the derivation tree defined by the operational semantics and builds the session types necessary. It follows more classic approaches of asynchronous session calculus. Contributions are observed for the `arrived` k and `typecase` k `of` $\{S_i : P_i\}_{i \in I}$ constructs.

- A runtime calculus is used to analyse runtime constructs such as endpoints. These constructs are called runtime since they are present at a runtime state of a specification. The important property about the runtime system is that it takes into account the consistency that must hold between the typed session interactions. For this purpose we define a *composition* discipline and a Session Type Remainder algebra. Runtime typing system completes the definition of the calculus and the session type typing system.

7.1.4 Programs and Typing

- Through example the power of the calculus is studied under a practical approach. For examples purposes we consider a core web server implementation as a first example and we continued by extending the server and implementing it on different architectures. This server is a simple web server that accepts requests from clients and responds by sending a file. As a second example we added interaction of the server with a cache process in order to get the requested file. This introduces the handling of multiple events in the application.

- The third example is implementation of the cache server in staged event driven architecture. It is interesting to see these applications implemented in a theoretical framework. The implementation seems easy and straightforward. Event interaction between processes that compose the server is done using session endpoints described by recursive session types. The second SEDA and final example is a more complex example that implements a mail server. We can see a network of SEDA components and the use of different events through this example.

- In all examples we have the same mechanism for accepting requests (events/sessions) from clients. A new session it is initiated and stored in a private session endpoint for later processing. The processing is done by retrieving an event context from a session queue and process

it according to its type. When we are waiting the response for an event we store the event context back to the queue and proceed with the next event context.

- Typing reveals even more interesting properties about this kind of implementations and the calculus in general. The specification that handles retrieving and storing of event contexts forms equations about session types that can only be solved using recursive session type structures. This is not at all surprising because of the continuous interaction with the storing/retrieving mediums.

- The session set type appears in the typing. Session set types are connected to the subtyping relation so we can also see a necessary application of the subtyping rule during the application typing.

- SEDA implementation present the above observation and also present recursive typing in the endpoints used for communication between application components/processes. For exchanging data between processes we do not create a new session for each new data so the recursion is expected due to the continuous interaction between components.

7.1.5 Soundness Properties

- The system was studied for the theoretical property of soundness. This is a rather extensive study on soundness and all other related aspects. We needed to defined notions, state and prove lemmas useful for proving soundness and finally we formed and proved Subject Reduction and Type Safety. Finally there is a discussion about deadlocks not predicted by the current typing system.

- We began the studying by defining a well formed typings. Intuitively the well formness of a typing shows communication consistency for a program. It says that every session has its typed endpoint and each communication is consistent. This definition is used in Subject Reduction to show that we need a program with well formed typing in order to have "'good"' execution of the program.

- The next definition is a relation that orders typings. This relation has as basis the reduction on the session prefixes and the typing that come up from this reduction. Intuitively a typing Δ' is ordered to another typing Δ if it contains what we expect after a session reduction with respect to typing Δ.

- We formed some basic lemmas that are used for soundness proving. *Weakening, strengthening, substitution, subject congruence* and a lemma that studies environment properties. This are studied for static and runtime typings. Weakening and strengthening are used in general where needed to prove a deduction. Substitution is important when substituting values through the reduction. Subject congruence is used when using congruence in the reductions.

- Full prove is given for all the lemmas. Weakening and strengthening are done by structural induction on the typing trees (static and runtime). The cases of the induction follow an identical pattern towards the truth of the lemma preposition. Substitution is also done by structural induction on the typing trees. Here we have different approaches for each case but generally we have an easy straightforward prove for every case. Subject congruence and the environment lemma do not use induction rather than other standard proof techniques.

- The Subject Reduction theorem states that a program with well formed typing that can be reduced can always reduce to a system that its typing is ordered with the well formed initial typing. It is proved by structural induction on the tree defined by the operational semantics.

- A corollary that generalizes Subject reduction follows and states that each derivation of a program with well formed typing has a well formed typing. This corollary it is helpful to prove Type Safety but it also shows the basic intuition for soundness. Well formed programs

derive well form programs. The corollary is proven by induction on the transition length of the derivation.

- An *error process* is a process that it is not desired to happen when we have a typable program. The typing system should be able to catch such cases. We define these processes one by one and this definition is useful to form the Type Safety theorem.

- Type Safety says that a well formed program cannot be reduced to an error process. Someone can argue that Type Safety is a corollary of Subject Reduction. The proof is shown by contradiction. We assume that well formed processed reduce to error processes and we show contradiction when error process are not well formed. What it remains is to show that an error process is either untypable or it doesn't have a well formed typing.

7.1.6 Deadlock Avoidance

- The discussion closes with some situations not predicted by the current typing system. A deadlock scenario is presented and argued in comparison with deadlock scenarios of other session type systems. A strategy for avoidance of this situation is given. We also prove the correctness of the strategy.

- In summary the project has met the requirements for a full theoretical description and the study of soundness of an event driven session type formalism. Everything is presented in a logical order following the pattern of other studies around the specification and soundness of a typed process algebra. We hope that this work will play an important role in the further study of concurrency using session types.

7.2 Related Work - Comparison

Having presented the Event Driven Session Type Discipline we can now make a comparison between this system and the systems presented in the background literature sections. For further information the reader can read chapters 2 and 3 where the comparing systems are introduced.

First we would like to see the differences with the classic session type theory [15],[25]. The major difference here is that communication uses data structures such as endpoints to become asynchronous. The classic communication is based on the handshake basis of the π calculus. The definition of events comes from asynchrony so asynchrony should be used in this model. We use asynchrony to establish sessions, to communicate and to store event contexts. If we try to implement the former and the latter without asynchrony then it seems that the result would be a very complicated solution. So asynchronous structures come natural in the specification of a session type event driven model and simplifies the entire discipline.

Apart from asynchrony the rest of the algebra is an extension of the classic session type algebra to a point that it can support events. Of course here the extensions have mostly to do with asynchrony. There are many similarities in the typing systems. The differences lie in the set session types that are define to handle events and the subtyping relation. A major extension is the runtime typing system that considers runtime entities such as endpoints. Some combination operators have a more complex definition and also we need to define a small session algebra which it does not exist in the clasic theory. If we see the proof of soundness this algebra requires more specific and more complicated definitions such as the *ordering* relation. The theorems follow a similar pattern with the classic approach on session types. Finally there is the discussion about deadlock and deadlock avoidance. Having the asynchrony and the `arrived` construct as a tool we can tackle deadlock situations. As we argue these deadlock situations can happen in the original session types algebra.

A more realistic comparison would be the comparison with the Higher Order session discipline [20]. This work has a very similar technical background with the current work. The higher order algebra definitions, type system and soundness follow the same pattern. The study of soundness in most cases follows the same definitions such as the *ordering* relation.

The major differences are that one algebra has to deal with events and the other has to deal with higher order computation. Higher order computation is handled by a ybrid system of π and λ calculus in the higher order case. A complete theory is developed around higher order and it is embedded through the definitions of the system. In the event based discipline events are handled through explicit dynamic typechecking, the subtyping discipline and the matching definition. Again these notions are embedded into the session type event driven system.

The event session type algebra and the typing system is an extended asynchronous algebra that uses endpoints. The new ideas presented here is the representation of events as sessions and the extension considers the event handling. The constructs introduced here are `typecase` k `of` $\{\}$, `arrived` and from the types point of view the set session types, a definition of subtyping that considers set session types and a matching tree that matches session types. Also the asynchronous session initiation plays an important role to the algebra. The typecase construct is a construct that considers types when its reduction happens. We can see runtime checking of types in the semantic rule that describes `typecase` k `of` $\{\}$. The matching done follows a tree on the subtyping relation, so all types must be known at runtime. Runtime type checking cannot be avoided when it comes to this algebra. When the execution cannot be predicted by the typecase construct the process simply gets stuck indicating the error. But a properly statically typechecked concurrent program cannot get stuck according to the type system soundness. The typecase construct is presented in [1], [2].

We can see a the obvious difference in the system described in [7]. This system handles runtime typing through the raise of exceptions and exception handler processes. This is a completely different approach. The choice of the typecase construct was a design choice that was made bearing in mind that a developer wants to have explicit control over the type of events that can happen. The development of an event based system that handles runtime checking with exception handlers is possible.

Finally we would like to compare our algebra with the hybrid event driven model presented in [17]. Note that this is a comparison between a theoretic model and an actual implementation. The comparison will happen on the principles of the two systems. First we would like to stress out that both models are type safe. Session types are type safe by definition while the Haskell library enforces type checking. The two approaches present similarities on scheduling discipline. Scheduling is done on input output of the computation in every concurrent entity. The programming idea in our algebra has the same principle. The context in [17] is done by using concurrent threads, a way opposite to our approach that does not spawn new threads. Also our algebra is on a pure event driven basis while the Haskell library is a hybrid system.

7.3 Future Work

Some directions about possible future work can be given with the end of the whole discussion.

The most promising direction is that of the implementation of this system into an actual programming framework. A work for implementing session types in Java can be found in [14]. This framework defines session constructs and extends Java so we can develop session type programs. The choice of Java was made due to the portability and the wide spread of the Java platform. SJ framework extends the Java compiler and provides a library for session handling. The implementation is still on a research stage.

In this implementation we have the sending and receiving between processes to be structured on socket entities called session sockets. The communication is based on protocols declared in the program. A communication sequence that involves send, receive, selection, branching and loop constructs must be done according to the protocol implemented. SJ framework develops constructs to handle control loops and branch structures between parallel Java Virtual Machines. The idea is that the control flow of one process is controlled by the other process. This way the select, branch structures are used in a practical way. When initiating a session we have runtime typing to check that the protocols used by each process agree for communication. An interesting part of the framework is the implementation of delegation. Delegation is a rather hard problem to solve due

to the lower level connections that must occur.

Based on this work we can continue developing the SJ framework by providing new constructs to be able to handle events as described theoretically by the present system. We believe that a practical approach on the subject will reveal even more of the possibilities or limitations for this model.

It is interesting to see how we can translate meanings around events used in this system to control structures that can be used for real programming in Java. We can see from [14] that the select - branch structure has been translated into higher level constructs to support a real programming language like Java. The interesting constructs in this algebra is `typecase of` {} and `arrived` . Also the notions of asynchronous session creation present great interest and the handling of subtyping. Java supports subtyping and polymorphism so this makes the case of handling event `typecase of` {} and subtyping even more interesting.

After finishing with the implementation, a standard way to evaluate these constructs is by developing a large event based application. This application should be stress tested in comparison with similar applications. This is a classic approach to study the performance of the system in contrast with other wide spread efficient applications.

A good example to implement would be a web server. There is a lot of debate how web servers should be implemented. The implementation directions are thread based and event based servers. A web server in this model would obviously serve the second approach. We are not in a position to predict the performance of such a web server at the moment, but a possible implementation seems a promising idea.

Different frameworks are also evaluated using benchmarks. We could get a classic benchmark for comparing concurrent programs, change it so it can be supported by a possible implementation of the event driven session type model and run the benchmark to get results. This is another way to evaluate these ideas.

A study of how session types constructs can be abstracted to programming language constructs can be found in [9]. This is a work that incorporates session types in a theoretic programming model closer to a programming language. This core language is called multi-threaded object oriented language MOOSE. This work presents a complete definition, type system and soundness for the system developed. It is an object oriented core language where SJ was based for implementation. We would like to see constructs for event driven session types to be abstracted in an object oriented level. This can serve as a basis for an implementation at a later stage. A more ambitious work would be to study the expressiveness of these constructs in contrast with the expressiveness of session types. This can tell us the power of a possible implementation of session type.

Another direction is to see the event discipline described here as a part of a more general session type system such as multiparty session types [13]. Multiparty session types allow more than two entities to be part of a communication making the algebra even more expressive. Problems that cannot be solved with the classic session type theory can be solved with multiparty session types. As concluded through the entire project events are a general model to describe concurrency, so we can easily extend multiparty session types to include an event driven programming model.

From the whole discussion around events and the possible models that sessions can describe we can develop session type models that model different approaches on the event driven models. Hybrid event systems seem very promising as far as efficiency and program developing. We would like to see session type disciplines extended with construct that describe hybrid models. Thread spawning is very trivial in π calculus, but the cooperation of processes to handle event driven programming as presented through the paper and through the literature seems to be a challenging work.

A very important notion in the event driven session type system is the way context and scheduling is handled. In order to have these two properties we use session endpoints. These endpoints are first in first out queues that result in a round robin scheduling. As we know from the operating system theory there are many approaches for context and scheduling that concern many different data structures. If we could define such data structures in a theoretic level then we can have different parameters on the entire algebra. On the other hand since this is a theoretical model the choice

of a data structure has no affect on the final conclusions we obtain by the study of properties such as soundness. The affect would come on a more practical model of this idea.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *J. Funct. Program.*, 5(1):111–130, 1995.

[3] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference table of contents*, pages 289 – 302, 2002.

[4] HP. Barendregt. *The lambda calculus: its syntax and semantics*. Elsevier, 1984.

[5] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 473–492. Springer, 2008.

[6] G. Boudol. Asynchrony and the pi-calculus, 1992. INRIA Res. Report 1702.

[7] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[8] Ryan Cunningham and Eddie Kohler. Making events less slippery with eel. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[9] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Inf. Comput.*, 207(5):595–641, 2009.

[10] Kevin Donnelly and Matthew Fluet. Transactional events. *J. Funct. Program.*, 18(5-6):649–706, 2008.

[11] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional events for ML. In *ICFP*, pages 103–114. ACM, 2008.

[12] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *TCS*, 410(2-3):202–220, 2009.

[13] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2008. ACM.

[14] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Honda K, VT Vasconcelos, and M Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP'98) at the Joint European Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1381, pages 122–138, 1998.

[16] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007.

[17] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, 2007.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[19] R Milner, J Parrow, and D Walker. A calculus of mobile processes. *Information and Computation*, 100:1 – 40, 1992.

[20] D Mostrous and N Yoshida. Session-based communication optimisation for higher-order mobile processes. In *Typed Lambda Calculi and Applications (TLCA09)*, volume 5608, pages 203–218, 2009.

[21] `http://www.mcs.anl.gov/research/projects/mpi/`.

[22] John Ousterhout. Why threads is a bad idea, 1996.

[23] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, volume 9, 2003.

[24] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP01)*, volume 35, pages 230–245, 2001.

[25] N. Yoshida and VT. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: *Two Systems for Higher Order Communication*. In *Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006)*, volume 171, pages 73 – 93, 2007.

List of Figures