Maria Luísa Sobreira Gouveia Lourenço

Type Inference for Conversation Types

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Lisboa, 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Type Inference for Conversation Types

Maria Luísa Sobreira Gouveia Lourenço

1º Semestre de 2009/10
22 de Fevereiro de 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Type Inference for Conversation Types

Maria Luísa Sobreira Gouveia Lourenço

Orientador: Prof. Doutor Luís Caires

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

1º Semestre de 2009/10
22 de Fevereiro de 2010

Acknowledgements

I would like to thank Professor Luís Caires for his support and guidance throughout the execution of this work. I thank my colleagues Bernardo Toninho and Mário Pires for their comments and suggestions of early stages of this work, João Martins for his suggestions and comments to improve some chapters of this document, and Hugo Vieira for his support.

I also want to thank my family (including my dog) and Ricardo Silva for their support and for their tolerance regarding my irritability and ill-temper when I am under stress.

This work was fully supported by a research grant from EU Project SENSORIA.

Resumo

Esta dissertação trata o problema de inferência de tipos no contexto de *tipos de conversação*, propondo e implementando um algoritmo de inferência de tipos. Este problema é interessante de abordar se tivermos em conta que as aplicações orientadas a serviços podem ter protocolos de uso muito ricos e complexos, requerendo assim ao programador que este anote todas as invocações de serviços com um tipo correspondente ao seu papel num protocolo, o que tornaria o desenvolvimento de tais aplicações nada prático. Assim sendo, libertar o programador de tal tarefa através da inferência de tipos que descrevem tais protocolos é algo bastante desejável, não só porque é enfadonho e difícil de fazer tais anotações mas também porque reduz a ocorrência de erros durante o desenvolvimento de sistemas reais e complexos.

Embora haja muito trabalho feito relacionado com *tipos de sessão* e inferência de tipos no contexto de sessões binárias, trabalhos relacionados com *conversações múltiplos participantes* ainda é escasso embora hajam algumas propostas relacionadas com conversações baseadas em multi-sessões (i.e. as interacções acontecem através de canais partilhados que são distribuídos, na altura da invocação, a todos os participantes).

A nossa abordagem é baseada no Conversation Calculus, um cálculo de processos que modela primitivas de serviços baseado em pontos de acesso de conversação, aonde todas as interacções acontecem. De modo a testar o nosso algoritmo de inferência de tipos, desenhamos e implementamos uma linguagem de programação distribuída e baseada no Conversation Calculus como *proof-of-concept*.

Finalmente, mostramos que o nosso algoritmo é coerente, completo, decidível e que retorna sempre um tipo principal.

Palavras-chave: Computação Orientada a Serviços, Sistemas de Tipos, Inferência de Tipos, Conversações, Tipos de Conversação

Abstract

This dissertation tackles the problem of type inference for *conversation types* by devising and implementing a type inference algorithm. This is an interesting issue to address if we take into account that service-oriented applications can have very rich and complex protocols of services' usage, thus requiring the programmer to annotate every service invocation with a type corresponding to his role in a protocol, which would make the development of such applications quite unpractical. Therefore, freeing the programmer from that task, by having inference of types that describe such protocols, is quite desirable not only because it is cumbersome and tedious to do such annotations but also because it reduces the occurrences of errors when developing real complex systems.

While there is several work done related to *session types* and type inference in the context of binary sessions, work regarding *multiparty conversations* is still lacking even though there are some proposals related to multi-session conversations(*i.e.* interactions happen through shared channels that are distributed at service invocation time to all participants).

Our approach is based on Conversation Calculus, a process calculus that models services' primitives based on conversations access point where all the interactions of a conversation take place. In order to test our type inference algorithm we designed and implemented a prototype of a *proof-of-concept* distributed programming language based on Conversation Calculus.

Finally, we show that our type inference algorithm is sound, complete, decidable and that it always returns a principal typing.

Keywords: Service-Oriented Computing, Type Systems, Type Inference, Conversations, Conversation Types

Contents

List of Figures

1. Introduction

Distributed computing has grown in use in the past years to increase performance of heavy computations (*e.g.* scientific problems, collaborative spaces, etc) as well as to provide services for public use (*e.g. e-commerce*), through distribution, parallelism and concurrency of entities representing computations.

Besides that, and also taking into account that interactions in distributed systems are message-based, software for distributed computing is prone to new types of errors. For instance, a peer can, wrongly, act upon messages that were received out of order or it can violate a global view of an interaction protocol in the system. For this reason, it is important and crucial to have tools for static analysis of distributed software and, in particular, of service-oriented software.

This is an interesting issue to address specially since there is a lack of practical tools or full fledge programming languages equipped with proper static analysis to verify that such errors do not occur during software execution.

In the last years, due to the increasing popularity of the Internet, services [36] have become a widespread approach to implement *e-business* and distributed applications. The choice to make use of services to achieve distributed computing seems reasonable, if not obvious, if we observe some important characteristics of services. First of, services are independent of their executing platform (*loosely-coupled* [20]) thus rendering them useful for interoperability of applications in a distributed system; secondly, they can be *published* so as to allow being *discovered* by other services as well as be *orchestrated* together to create more complex systems; lastly, services can invoke other services, in a seamless fashion to the client, to complete part of their work (*delegation*).

1.1 Objectives of the work

This dissertation's goal consists in designing and implementing a typechecking algorithm with type inference for conversation types. To do so, we propose and implement a prototype for a *proof-of-concept* distributed programming language based on Conversation Calculus.

Why is it important to devise a type inference algorithm in the context of conversation types? Taking into account that service-oriented applications can have very rich and complex protocols of services' usage, then requiring the programmer to annotate every service invocation with the corresponding part in a protocol would make the development of such applications quite unpractical. Therefore, freeing the programmer from that task, by having inference of types that describe such protocols, is quite desirable not only because it is cumbersome and tedious to do such annotations but also because it reduces the occurrences of errors (the programmer doesn't have to worry about giving the right type to an invocation) when creating real complex systems. Part of this dissertation's results are presented in [23], accepted for publication in the pre-proceedings of Programming Language Approaches to Concurrency and Communication-cEntric Software 2010 (PLACES'10).

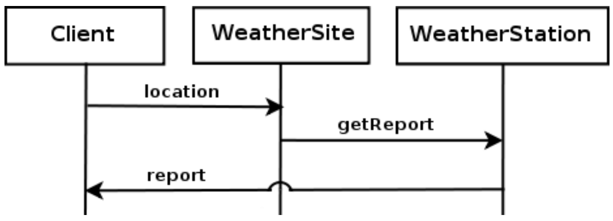


Figure 1.1: Toy example's messages sequence.

1.2 Computational Abstractions

As previously stated, we can achieve distributed computing by making use of concurrency, parallelism and distribution with entities that represent computations. These entities can be objects, components or services.

Objects describe computations through a set of fields (data) that can only be manipulated through a set of methods (functions) defined in the object, this way a client can only invoke methods to perform computations on the data defined inside the object or passed as parameters.

Components, however, can be seen as an encapsulation of objects with a set of available ports to connect to other components, this way allowing the construction of more complex systems. The interaction with a client is made through the plug of the component itself.

On the other hand, services add to components a higher level of abstraction by offering a specific functionality to the client that, internally, can be achieved through the composition of other services or partially delegated to another service. So a service can be implemented through a component or even an object, what matters is that the interface of a service is well defined through standard description languages and can be provided by different physical sites thus allowing interoperability between different services' implementations on heterogeneous platforms. The interactions found in services are structured and usually between two or more peers, this is known as a *conversation*.

1.3 The Weather Forecast Service in our language

We now present a toy example using our language's syntax to illustrate some concepts. To simplify we will omit remote types' declarations for now. This example models a weather forecast service that, when invoked, awaits for the client's location, asks the associated weather station to join the on-going conversation (Client−WeatherSite) and requests a weather report. After the weather station joins the conversation, it will generate a weather report and send it directly to the client. Figure 1.1 describes the messages sequence of our example whilst Figure 1.2 shows the code for both services' definitions on their respective sites.

Notice that when there is a service invocation through the **join** primitive, we are in fact adding another partner to an on-going conversation instead of creating a new conversation,

```
site #WeatherStation {
   def #weatherReport as {
      receive(#getReport);
      send(#report, generatedReport);
   }
};;
```

```
site #WeatherSite {
   def #forecastWeather as {
      val loc = receive(#location);
      join #weatherReport in #WeatherStation as {
         send(#getReport);
      }
   }
};;
```

Figure 1.2: Services and Sites declarations' code.

this is called *partial delegation* since WeatherSite doesn't lose its capability to interact in the current conversation. In this toy example we end up with a *conversation* between the Client, WeatherSite and WeatherStation which enables the weather station to communicate directly to the client. The client's code (Figure 1.3) is simply the invocation of the weatherReport service of WeatherSite followed by the client's communication of where it is located to the invoked service and, finally, it awaits for the weather report, unaware that it is another service responding to its request.

```
invoke #forecastWeather in #WeatherSite as {
   send(#location, my_location);
   val my_weather_report = receive(#report)
};;
```

Figure 1.3: Client's code.

Going back to our example's message sequence, we can perceive it as a global view of the protocol the conversation between all partners must comply to. In particular, each participant of the conversation must comply with part of the protocol. This protocol compliance verification also enables us to verify the correction of messages exchanges w.r.t. their ordering thus assuring the good-behaviour of each partner. Thus we need specific static analysis techniques for service-oriented software.

In particular, we can use *conversation types* to describe these behaviours. A conversation type is an association between a site's name and its services, $[s](B)$ where s is a service name and B a behavioural type describing the interactions that take place within the service. Taking a look at our toy example with conversation types we obtain the following:

Client site :
#WeatherSite: [#forecastWeather](#location?(String);#report!(String))

WeatherSite site :
  #WeatherStation: [#weatherReport](#getReport?(String);#report!(String))
  #WeatherSite: [#forecastWeather](#location?(String);#getReport[tau](String);#report!(String))

WeatherStation site :
  #WeatherStation: [#weatherReport](#getReport?(String);#report!(String))

The type for the #weatherReport service in the #WeatherStation site states the service's behaviour when invoked: first it awaits for a communication in label #getReport and then sends through label #report a string.

For the client's site we have a type describing the local behaviour of the invoked service #forecastWeather which says the client will first send through label #location a string and then await a string in label #report.

Finally, on the #WeatherSite site the type for the service #forecastWeather will state that a string is expected to be received through label #location, an internal interaction occurs via label #getReport, and a string is sent through #report. The internal interaction represents the communication done with service #weatherReport, located at #WeatherStation, to obtain the weather report while what follows from this interaction, #report!(String), is the residual behavioural type from the service #weatherReport (due to joining the conversation).

We can typecheck that the client invocation complies with the expected behaviour by verifying its duality w.r.t. the service's behavioural type, *i.e.*, in this case the client sends a string through #location while the service awaits a string via #location and, after performing an internal interaction, the service sends through #report a string while the client awaits a string via #report.

1.4 Software Analysis

To reason about the correctness or behaviour of software, an appropriate mathematical model is necessary. In this sense, process calculi [28] offer a powerful tool to reason about the interactions between concurrent processes by means of a simple modelling language with algebraic properties that enables us to verify equivalence of processes. Another method for program analysis is the definition of a type system [32, 7] that allows us to specify the good behaviour of our programs through a set of inference rules that programs must comply with.

A variety of process calculi emerged to reason about distributed computing, from broader calculi to represent concurrent and distributed systems to more specific calculi to model service-oriented systems. In particular, service-oriented calculi are based on two approaches: multi-session based interactions and conversation based interactions. The difference consists in how the interactions happening in a *conversation* are modelled: the multi-session based approach represents interactions through shared channels that are distributed at service invocation time to all participants (that must be know beforehand), thus creating binary sessions between each partner; the conversation based approach describes the interactions through a conversation access point where all the interactions of a service takes place (notice no multicast or knowledge of all partners are necessary).

Type systems for concurrent and distributed systems have evolved in the past years, first by reasoning about what kind of values one expects from communication channels (since concurrency systems usually have message-based communication) to describing the behaviour a program has with respect to what messages it receives or sends. More recently, type theory was developed for service-oriented systems based on the mentioned two approaches: for multi-session based systems, a global description of all the interactions between conversations is given (much like a choreographic description of services) through a global type whilst a local type describes each partner's local behaviour; for conversation based systems, an unification of a global and local description of behaviour is achieved by a conversation type to each partner that can be unified into a more general conversation type describing the expected contract for that service.

1.5 Contributions

This work's contributions are:

- Design and implementation of a proof of concept programming language based on Conversation Calculus;
- Implementation of a prototype of the associated distributed runtime system;
- Design and implementation of a typing inference algorithm;
- Proof of completeness, correctness and decidability for the algorithm and proof that it always returns a principal typing;
- Validating examples.

1.6 Document Structure

This document is structured as follows: Chapter 2 establishes the context of this thesis's work by first describing Service-Oriented computing and services along with their relevance nowadays for distributed computing; then an introduction of process calculi and type systems is given by a brief survey of the important and relevant work done in the area for service-oriented computation. In Chapter 3 we present an overview of our prototype language and distributed runtime support, presenting some examples of its usage. We follow with the formalisation of our type system and typechecking algorithm in Chapter 4, where we also show the main results of this dissertation. Chapter 5 makes a brief discussion about the implementation of this work. Finally, Chapter 6 draws some concluding remarks about the presented work, taking into consideration the initial goals and what we obtained in the end, and describes some future work.

2 . Technical Background

In this chapter we introduce related background by first explaining some Service Oriented computing concepts and technologies, then we present a couple of software analysis methods, process calculi and type systems, and show their evolution up to the analysis of Service Oriented software. In particular, we present the models and techniques from which we will be basing our work on.

2.1 Service Oriented Computing

Service Oriented Computing (SOC) [36, 1] is a recent paradigm that has been widely used in distributed systems to achieve distributed computing (whether to solve scientific or business problems) as well as to provide utilities for public use (*e.g. e-banking* over the Internet). This paradigm is centred around the concept of service as a computational process (specifically a container of code) that can be published, discovered and orchestrated with other services. The use of standards in protocols and description languages (that state the service's interface as well as its intended behaviour) makes it possible to put together applications that would otherwise be incompatible due to their execution platforms. Thus, services are autonomous regarding the platform where they're executed, *i.e. loosely-coupled* [20], enhancing the interoperability of applications in a distributed system environment. Furthermore, we can create more complex services through the composition of services, known as orchestration of services. A service can also delegate part of its work to another service (for *e.g.* a Seller service can delegate the shipping of an order to a Shipper service), this is called process delegation.

These characteristics make services a popular approach for distributed applications and, in particular, for Web applications where we deal with a huge, heterogeneous network and where e-business has proliferated in the past years. It is not a surprise then that many models and programming languages have been developed to aid in the construction of service oriented software.

This has called for some standardisation by the World Wide Web Consortium (W3C), otherwise it would make integration of web services from different technologies harder or impossible.

A Web service's language use the Extensible Markup Language (XML) to define messages that are exchanged using the Simple Object Access Protocol (SOAP). A web service's interface of possible operations is described using a specific description language, the Web Service Description Language (WSDL). This description language is only a prerequisite for automated client-side code generation that is offered by many Java and .NET frameworks. Frameworks such as Spring, Apache Axis2 and Apache CXF make no such demand.

There is also an architecture to define web services applications that uses the HTTP protocol for communication instead of SOAP. By doing so XML and WSDL are not required to define the service's messages and API, respectively. The main focus of this type of architecture is to

$$P \quad ::= \quad \mathbf{0} \mid P + Q \mid P|Q \mid a.P \mid P[b/a] \mid P\backslash L \mid A$$

Figure 2.1: CCS's syntax.

deal with stateful resources, rather than messages or operations. For more information about basic concepts in SOC or regarding most of the previously enunciated technologies, see [1].

In general, the problems that arise in communication based software are not new when considered in isolation. But in the context of Web Services, where we have multiparty communication along with loose-coupling of distributed systems as well as description based discovery of services, new issues arise that need to be discussed and analysed through new models with new concepts.

Software analysis is crucial to ensure applications comply to their specification and do not incur in certain types of errors during their execution, as well as to give us guarantees that certain properties will always hold (*e.g.* deadlock-free).

2.2 Process Calculi

Process calculi offer a powerful abstraction to real computational systems by means of a simple, yet expressive, modelling language for concurrent systems with algebraic laws that permits a process to be manipulated in an equational manner and, therefore, allow us to reason about process equivalence. The basic unit of a process calculus is a *process* which consists in an agent that can interact with other processes through communication using channels.

The abstraction a process calculus offers is necessary to reason about real complex systems where we need to prove that certain properties always hold (safety, liveness, etc) but it is not enough to ensure certain ill behaviours do not occur during the execution of a program. For this a type system [32, 7] is best suited because it provides a set of rules, called *typing rules*, that when successfully checked against a program can certify it won't incur in ill behaviour during its execution.

2.2.1 Calculus of Communicating Systems

Milner introduced one of the first process calculi, the Calculus of Communicating Systems (CCS) [27], which consists of a small set of operators, represented in Figure 2.1. First we have the *inaction* process **0** that models a process that can not perform any action. This construction alone isn't interesting but if we want to describe, for instance, a broken machine that warns the user of an abnormal error through a *beep* sound,

$$\overline{\texttt{beep}}.\mathbf{0}$$

then we can combine the inaction process with another constructor, the *action-prefixing* a.P, that states a process can perform action a and then proceed as process P. In this case, we're saying the broken machine makes a *beep* sound and then ceases to function (since it can no longer do any action). However, initially, the broken machine worked as intended and only after some time, for some reason, broke. This behaviour motivates our next operator, the *summation* P + Q, which indicates a process can either act as process P or as process Q. To be able to model recursive behaviour we need to present *process definitions*, which are identifiers, in the set of process identifiers A, that can be used to represent processes in our calculus. For example,

$$\texttt{Machine = work.Machine + }\overline{\texttt{beep}}\texttt{.}\mathbf{0}$$

here we're defining a process named `Machine` that behaves like intended through the action `work` until it breaks down and stops functioning. The summation captures the non-determinism of the machine's behaviour stating that at some point it may stop working. To turn our example into a more realistic machine we need to add a switch to turn it on/off for anyone that wishes to use the machine. We then introduce our next constructor, the *parallel composition* P|Q, that allows us to compose more complex systems by combining processes P and Q. Furthermore, this composition executes the composed processes in parallel hence opening the possibility for communication and concurrency. Communication is achieved by the use of actions, a, that can synchronise with their complementary action, $\overline{a}$, becoming then an internal action, τ. An action can then be an input of the form a(x), where x is a bounded variable that is instantiated when a synchronisation occurs on action a, or an output of the form $\overline{a}$(v) where v is the transmitted value when a synchronisation occurs. To model a more realistic machine we have:

$$\begin{array}{l}
\texttt{Machine = onSwitch.Work} \\
\texttt{Work = }\overline{\texttt{work}}\texttt{.Work + offSwitch.Machine + }\overline{\texttt{beep}}\texttt{.}\mathbf{0} \\
\texttt{User = }\overline{\texttt{onSwitch}}\texttt{.}\mathbf{0} \\
\texttt{System = Machine | User}
\end{array}$$

where we compose a system with an user and a machine. The user simply interacts with the machine by turning it on while the machine awaits for the "on switch". When turned on, the machine can display the following behaviour: it either works as intended or awaits for the switch to be turned off and awaits, again, for the "on switch" or, finally, it breaks down after emitting a beep sound. On the other hand, this system has a flaw when concerning who can utilise it. Let's say the machine, for security reasons, is restricted in the sense that only our user `User` should be able to operate with it. Then if we had another user he shouldn't be allowed to use the "on switch" of the machine by communicating through action `onSwitch` but, as it is now, he can in fact do so thus breaching the system's security. To solve this we introduce the *restriction* operator P\L which states process behaves like P except that it loses the capability to communicate through any action in the set of actions L. We can then restrict our system in the action `onSwitch` rendering useless its capability to communicate outside the system, as intended.

$$\begin{array}{lll} P, Q & ::= & \mathbf{0} \mid \sum_{i \in I} \alpha_i.P_i \mid P|Q \mid (\nu n)P \mid !P \\ \alpha & ::= & \overline{\alpha}\text{<v>} \mid \alpha(x) \mid \tau \end{array}$$

Figure 2.2: π-calculus' syntax.

$\texttt{Machine2 = onButton.Work2}$
$\texttt{Work2 = }\overline{\texttt{work}}\texttt{.Work + offButton.Machine + }\overline{\texttt{beep}}\texttt{.}\mathbf{0}$

Lastly, let's imagine we have another machine `User` can access to that behaves in the same way as `Machine` but instead of an "on switch" it has an "on button". The interaction `User` would have with this new machine, let's call it `Machine2`, is the same he would have with the first machine: turning it on. But this new machine doesn't have a "switch" so the user can no longer perform the action $\overline{\texttt{onSwitch}}$ unless we state, somehow, that performing an "on switch" action is the same as an "on button" action w.r.t. powering up the machine. This is possible with the *labelling* operator P[b/a] where all actions a are relabelled to b. In our example we can say `Machine2[onButton/onSwitch]` and `Work2[offButton/offSwitch]` or `User[onSwitch/onButton]`, either way makes the intended interactions possible. Furthermore, we could define `Work2` as the relabelling of all actions `offSwitch` to actions `offButton` (`Work[offSwicth/offButton]`).

As we have seen, this small set of primitives is expressive enough to model concurrent processes that communicate with each other. However, the communication itself is abstracted as synchronisation of processes through a communication channel instead of data exchange *per se*. Thus CCS lacks the expressive power to represent dynamic reconfiguration of the system and name passing. For instance, if we wish to model a HTTP server that accepts requests and launches a thread per client request to initiate a session, we would have trouble to represent the mobility of the initial link between the client and the HTTP server to a new thread in the HTTP server. In fact, we can not fully represent this mobility since there's no way of generating new actions to represent this new link as well as pass it to the thread.

2.2.2 π-calculus

This has motivated further work that later progressed to a new and powerful process calculus, the π-calculus [28], that is able to represent such systems as the one presented above. The π-calculus, also a work of Milner, is an extension of the CCS where a new operator is added to create *fresh* names and values can be passed through channels. Looking at the π-calculus' syntax, Figure 2.2, we can verify the similarities to the CCS's syntax. We have an *inaction* process, a *parallel composition* primitive, a *guarded summation* with $i \in \mathbb{N}$ prefixed processes (notice we can obtain the prefix process of CCS's syntax when i is 1), and the *replication* operator which has the same functionality that process definitions had in CCS. The novelty of this work is in the new operator (νn)P that extends CCS's restriction operator by generating a fresh name n whose scope is process P. Furthermore, we can output these restricted fresh names to another process and, by doing so, extend its scope (this is called *scope extrusion* in the literature). This offers an increase of expressive power to model more realistic systems where we have mobility of names. For example, we can now represent the previous HTTP server - Client system (we use the symbol $\overset{\text{def}}{=}$ to define alias for process definitions since π-calculus does not have an operator for it).

$$
\begin{array}{l}
\texttt{HTTPServer} \overset{\text{def}}{=} \texttt{handshake().}(\nu\texttt{ch})\overline{\texttt{init}}\texttt{<ch>.}\overline{\texttt{reply}}\texttt{<ch>} \\
\texttt{Thread} \overset{\text{def}}{=} \texttt{init(x).x(cmd)} \\
\texttt{Client} \overset{\text{def}}{=} \overline{\texttt{handshake}}\texttt{<>.reply(x).}\overline{\texttt{x}}\texttt{<GET>} \\
\texttt{System} \overset{\text{def}}{=} \texttt{!(HTTPServer | Thread) | Client}
\end{array}
$$

The `HTTPServer` awaits for a handshake (represented as a communication on channel `handshake`) then communicates a freshly generated name through channels `init` and `reply`; `Thread` represents a thread that is awaiting on `init` for a channel from which commands will be transmitted; and `Client` initiates a handshake with the server through channel `handshake` then awaits for a reply that will contain a channel to which it will later send a "GET" request.

`System` represents the described system: a persistent HTTP Server that launches a thread per handshaked client. To be able to model the persistency of the server we use the replication operation whilst to ensure the condition "one thread per handshake" is satisfied, we compose the HTTP server with a process that models a thread (so in fact the replicated HTTP server is the composition of `HTTPServer` with `Thread`). The key idea of this example is to show the mobility of the name `ch` (modelling the link between a client and a thread from server) from `HTTPServer` to `Thread` and `Client`. This allows us to construct systems with dynamic reconfiguration of its processes' connectivities as well as transmission of names (that are unique).

2.2.3 Calculus for Multiparty Conversations

The interaction model found in SOC is sensible to (as well as structured around) its context. However communication primitives in π-calculus are unable to capture that kind of interaction. This has motivated the study of service-oriented formalisms that can incorporate richer techniques to reason about SOC. To further introduce the calculi, we now present some key concepts. A *conversation* is a structured, potentially concurrent and distributed interaction between two or more parties. The medium where a conversation holds is called a *conversation context*. A *session* (or *binary session*) is an instance of a conversation where two, and only two, parties are involved.

The notion of *session* as structured dyadic interactions was first described in Honda's work [16, 38] and is represented as a private channel (*session channel* for future reference) through which all communication within is made. In [17] (later revisited in [42]) Honda, Vasconcelos and Kubo proposed a set of primitives to structure communication-based concurrent programming based on Milner's $\pi - calculus$ [28] and their type discipline. We present a fragment of

$$\begin{array}{lcl} P, Q & ::= & \text{request a(k) in P} \mid \text{accept a(k) in P} \mid k![\tilde{e}];P \mid k?[\tilde{x}] \text{ in P} \\ & & \mid k \triangleleft l;P \mid k \triangleright \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\} \end{array}$$

Figure 2.3: Fragment of session π-calculus' syntax

the language used (Figure 2.3) to illustrate some of its expressive power.

Communication can only be made within a session so to initiate a binary session primitives request a(k)in P and accept a(k)in P are used together. The former sends a request for session initiation via name a and awaits for the generation of session channel k while the latter receives the request through name a and generates a fresh name k to be used as a session channel. Session channel k is then used for all the interactions happening in the initiated session.

Two primitives for communication are introduced: *label branching* and *delegation*. The first allows for a form of method invocation where a session channel can offer an interface of methods represented by processes and identified by *labels* ($k \triangleright \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\}$), the invocation is then made through a *label select* construct ($k \triangleleft l;P$). The latter represents a delegation of a session through channel passing to another process, thus enabling dynamic distribution of a session among processes. However, this delegation primitive has a total delegation semantics, meaning whoever delegates a session can no longer participate in it.

For instance, we can encode a session between a terminal of a computer and a user of the system. The computer offers three options once a user is logged on: to list its files; to copy a file; and to remove a file. To simplify we'll only encode the action the user will use, the listing of the files. In this example the user will initiate a session on a terminal and list its files through option `dir`. On the other hand, the terminal awaits for a connection and once established it offers three possible actions and acts accordingly.

```
User ≝ request console(my_session) in my_session![login, password];
    my_session◁dir: my_session?[list]
Terminal ≝ accept console(my_session) in my_session?[x, y];
    my_session▷ { dir : my_session![data] || cp : P || rm : Q }
```

In [2] the authors proposed a process calculus, Service Centered Calculus (SCC), with primitives for service definition, service invocation and service handling (but not service delegation). SCC was developed in the context of the EU Project SENSORIA [1]. Their main result was establishing a mathematical basis for structured interaction that is typically found in SOC that allows for formal reasoning about services by verification methods (*e.g.* type systems). SCC incorporates a notion of session that is established between the client-side (the invoker) and the server-side (the invoked) where communication is bi-directional and done through concretions and abstractions [28] (instead of using a single channel like in [17]). It has a mechanism for

[1]EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA), website: http://www.sensoria-ist.eu/

session termination that allows one of the sides to abort or terminate a session and communicate to its partner the decision who will, in turn, execute a specific handler to treat the abnormal termination of the session. SCC, however, does not support multiparty conversations.

A proceeding work from [17] was presented in [18] where Honda, Yoshida and Carbone extend the notion of *binary session* to *multiparty session* allowing for more than two participants in a conversation (or session). This is achieved by a multicast mechanism upon session initiation that distributes fresh channels (only to be used in this conversation) through all the participants. The calculus, however, does not permit a dynamic number of participants (these must be know beforehand upon service instantiation) thus limiting what we can express using this calculus.

Another calculus for multiparty sessions was introduced in [3] by Bruni *et al.* but based on SCC [2]. The notion of a multiparty session is here represented by session endpoints (instead of multicast of fresh channels found in [18]) through which participants can access the session. Thus a service invocation executes the service's process via the session endpoint of the invoker but only on the server side (this differs from [2] where service invocation would generate a fresh session on both sides). It is also possible to dynamically join a session through a *merge endpoint* that "merges" sessions endpoints, offering a common session endpoint as a result (hence both participants are now in the same conversation). This enables service delegation with a partial semantics, meaning the delegated process can proceed in the conversation afterwards.

Communication is separated into two levels: intra-session communication and intra-site communication. They differ in their scope, the former is only captured by participants of the session (that can be physically located at different sites), without risk of interference from other sessions' communication, whilst the latter is captured by processes in the same location (site) enabling local communication. Lastly, to model sites the authors offer a construct to associate a name to a process. Although in [18] there's no such construct, the notion of site is captured by the type system through a *located type*.

The previous models have limitations regarding their implementation of multiparty conversations in the sense that multiple participants do not evolve dynamically. Furthermore, those models tend to be complicated and hard to understand mostly due to their cumbersome syntax. This has motivated the development of a process calculus to address these issues, the Conversation Calculus, an extension of the π-calculus with context-sensitive communication primitives and access points to contexts (Figure 2.4). Our work is based on this process calculus. It was first introduced in [41] by Vieira, Caires and Seco and later refined in [4] by Vieira and Caires. In this calculus a conversation is the basic unit that can be accessed via a conversation access primitive (n ◂ [P]), similar to how a multiparty session's conversation is accessed through session endpoints in [3].

Here, just like in [3], partial delegation is possible and is achieved by using a special primitive, **this**(x), of the calculus to dynamically obtain the current conversation's identity (a name) and then send it to another process (π-calculus' scope extrusion) that will later on use it to access the conversation (via conversation access construct). This mechanism differs from the approach of [3] where delegation is done by merging two sessions together while here we pass

$$\begin{array}{lcl} P, Q & ::= & \mathbf{0} \mid P|Q \mid (\nu n)P \mid \mathbf{rec}\, X.P \mid X \mid n \blacktriangleleft [P] \mid \sum_{i \in I} \alpha_i.P_i \\ d & ::= & \uparrow \mid \downarrow \\ \alpha & ::= & l^{d}!(n) \mid l^{d}?(n) \mid \mathbf{this}(x) \end{array}$$

Figure 2.4: Conversation Calculus's Syntax

the "capability" to access a conversation.

Communication actions can either be $l^d!(n)$ for sending messages or $l^d?(n)$ for receiving messages. Thus, what defines a communication message is its direction d as well as its label l. There are two message directions: $\uparrow$ for interactions in the enclosing conversation and $\downarrow$ for interactions in the current conversation.

Regarding message labels, they are not names but free identifiers meaning they are not subject to fresh generation, restriction or binding. Only conversation names may be subject to binding and freshly generated via $(\nu n)P$. Labels are associated with a conversation in the sense that even though we can have the same labels on different conversations, they won't interfere with each other.

Communication done in the current conversation ($\downarrow$), essentially, can be seen as the intra-session communication found in [3] whilst communication done in the caller conversation ($\uparrow$) can be regarded, partially, as [3]'s intra-site communication except that only "goes up" one level.

Although there's no construct to represent *sites* in Conversation Calculus, they can be represented as a top level conversation from where further conversations would be initiated.

The Conversation Calculus is expressive enough to represent common primitives found in services such as service definition, service instantiation, conversation join (these three were initially constructs of the initial calculus in [41]) and persistent service definition:

$$\begin{array}{lcl} \mathbf{def}\ s \Rightarrow P & \overset{\text{def}}{=} & s?(x).x \blacktriangleleft [P] \\ \mathbf{new}\ n \cdot s \Leftarrow Q & \overset{\text{def}}{=} & (\nu c)(n \blacktriangleleft [s!(c)] \mid c \blacktriangleleft [Q]) \\ \mathbf{join}\ n \cdot s \Leftarrow Q & \overset{\text{def}}{=} & \mathbf{this}(x).(n \blacktriangleleft [s!(x)] \mid Q) \\ \star\mathbf{def}\ s \Rightarrow P & \overset{\text{def}}{=} & \mathbf{rec}\ X.s?(x).(X \mid x \blacktriangleleft [P]) \end{array}$$

We now present a typical interaction in some file-sharing P2P protocols where we often find a dedicated server whose only purpose is to index files (tracker for future references) and its owners and peers that can either request a file or serve a file to another peer. We can encode this interaction as a service-oriented computation in the following sense: a tracker provides a service to request an indexed file; and a peer provides a service to transfer a file to another peer.

In the following example, a peer that wants to download a file named "House M.D." sends a request to the tracker; in turn the tracker will consult its database and return one of the peers that has the said file; the peer that requested the file can now establish a connection with the owner of the file and request it directly; the owner fetches the file from its hardrive and starts sending it to the requesting peer; finally, when it finishes sending the file, the owner informs the tracker the existence of a new owner.

```
Tracker ◄ [ def getFile ⇒ name(x,z).
                           search↑!(x).
                           results↑?(y).
                           peer!(y).
                           done?().
                           addPeer↑!(z)
          | DB
          ]

Peer ◄ [ new Tracker · getFile ⇐ name!("House M.D.", Peer).
                                  peer?(x).
                                  join x · download ⇐ file!("House M.D.").
                                                      sending?(y)
       | def download ⇒ file?(x).
                        fetch↑!(x).
                        getData↑?(data).
                        sending!(data).
                        done!()
       | HD
       ]
```

When a peer wants to download a file, "House M.D." in this case, it invokes the `getFile` service available in the tracker, this instantiation is made through a freshly generated conversation context that is extruded to the tracker process (thus closing the conversation context). In this new context conversation, only the peer that is requesting the file (client for future reference) and the tracker can communicate.

The client enquires the tracker for the desired file and awaits a response with one of the peers that owns the file. Meanwhile, the tracker starts a search in its database (represented by process `DB`) by sending a request in the up direction ($\uparrow$). Notice that after it crosses a conversation context boundary, the direction of the message changes to the current conversation context ($\downarrow$).

After getting the results from its database, the tracker answers the client with the resulting peer (server for future reference) and awaits for an acknowledge from the server so it can add the client in its database.

When the client receives the server identity, it asks the server to join the current conversation context by instantiating the `download` service (provided by the server) with the current conversation context. Then the client requests the file, the server obtains the file's data from its hard drive (in a similar fashion that the tracker searched its database), sends the said data to the client and, since it joined the conversation context where the tracker is also participating, sends the client identity to the server.

2.3 Type Systems

We now present some basic notions of type systems and show the motivation for their use in software analysis through their evolution from the functional programming paradigm to the service-oriented paradigm. Furthermore, this section presents some of the techniques we will be using in our work to verify a program behaves properly during its execution. Except otherwise noticed, most basic type-related concepts explained in this section are covered in [7, 32]. A type system is a syntactic method that attributes to terms of a program a *type* describing what kind of value is computed in such way that assures the program will never misbehave during its execution. The most common method to describe a type system consists in defining a set of rules, called *typing rules*, to which a program must comply to. A typing rule is composed by a set of *typing judgments* of the form $\Gamma \vdash P : T$, the premises of the rule, and a final typing judgment, the conclusion, that holds when all the premises are satisfied:

$$\frac{\Gamma_1 \vdash P_1 : T_1 \qquad \dots \qquad \Gamma_n \vdash P_n : T_n}{\Gamma \vdash P : T}$$

A typing judgment $\Gamma \vdash P : T$ states that term P has type T under the typing environment Γ (where *typing assumptions* of the form $\mathtt{x}{:}\tau$ are kept, *i.e.* associations between variables and types). A term is said to be *well-typed* if a type can be *derived* from the typing rules.

The act of verifying a program complies with a given type system is known as *type checking* and can be done at compile-time (*static typing*) or at execution-time (*dynamic typing*). Static typing maps types to variables of the program and allows us to catch many errors earlier than in dynamic typing and, in addition, enables code optimisation. On the other hand, Dynamic typing adds more flexibility by mapping types to values and, therefore, are less conservative than the static typing. For example, the instruction

```
if <complex test> then 5 else <type error>
```

is rejected by static type checking (but not by the dynamic counterpart) even if the `<complex test>` always evaluates to `true` and the `else` branch is never executed. However, static type checking gives us more guarantees exactly because of its conservative nature: a program when accepted will always behave as intended for all possible instantiations. In our work we will focus on static type checking.

A key property of type systems is *soundness* (or *safety*) that gives us the guarantee “Well-typed programs never go wrong.”. The soundness of a type system can be proved if the following two properties are true: a well-typed term stays well-typed under evaluation rules (*type preservation*); and a well-typed term never gets “stuck”, *i.e.* either its a value or further transitions are possible (*progress*).

2.3.1 Types for the λ-calculus

To show how type systems work, we now take a look at a λ-calculus' type system [7], Figure 2.6. In λ-calculus (defined in Figure 2.5) we can define functions and apply functions. Functions are therefore first-class citizens, *i.e.* they can be passed as arguments or returned as results. For instance, we can represent a function that sums its two arguments as `λx.λy. x + y`.

$$\begin{array}{lll} \mathtt{P,\ Q} & ::= & \mathtt{x \mid P\ Q \mid \lambda x{:}\tau.P} \\ \tau,\ \upsilon & ::= & \tau \to \upsilon \mid \nu \qquad \nu \in \mathtt{\{Int,\ Bool,\ ...\ \}} \end{array}$$

Figure 2.5: λ-calculus syntax

$$\frac{}{\varnothing \vdash \diamond}\ (\text{ENV } \varnothing) \qquad \frac{\Gamma \vdash \tau \qquad x \notin dom(\Gamma)}{\Gamma,\ x : \tau \vdash \diamond}\ (\text{ENV X})$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \upsilon}{\Gamma \vdash \tau \to \upsilon}\ (\text{TYPE FUN}) \qquad \frac{\Gamma \vdash \diamond \qquad \nu \in \{Int, Bool, \ldots\}}{\Gamma \vdash\ \nu}\ (\text{TYPE BASIC})$$

$$\frac{\Gamma,\ x : \tau,\ \Gamma' \vdash \diamond}{\Gamma,\ x : \tau,\ \Gamma' \vdash\ x : \tau}\ (\text{VAL VAR}) \qquad \frac{\Gamma \vdash x : \tau \qquad \Gamma,\ x : \tau \vdash P : \upsilon}{\Gamma\ \vdash\ \lambda x : \tau.P : \tau \to \upsilon}\ (\text{VAL FUN})$$

$$\frac{\Gamma \vdash P : \tau \to \upsilon \qquad \Gamma \vdash Q : \tau}{\Gamma\ \vdash\ PQ : \upsilon}\ (\text{VAL APPL})$$

Figure 2.6: λ-calculus' typing rules

In this example we can see a desirable property we wish for the type system to ensure: only integers are applied to the function, or more generally, only values of the proper types are applied to a function. This is described in rule (VAL APPL) that states that if term P is a function of type $\tau \to \upsilon$ and argument Q has type τ then we can apply the function and its application will have type υ. This rule enforces, as intended, our example to only take integers as arguments and return an integer as a result.

To be able to derive these types we need to include some typical rules (that from now on are implicit in our examples): the axiom (ENV $\varnothing$) that says the empty environment $\varnothing$ is well-formed $\diamond$; the (ENV X) that allows us to extend an environment as long as variable x isn't already defined in the environment; and (TYPE BASIC) stating all basic types are well-formed under a well-formed environment.

A type system can then be seen as a syntactic method to prove properties we wish our programs to hold as well as to enforce well-behaviour of programs.

2.3.2 Types for the π-calculus

Taking the polyadic π-calculus as an example we would like to assure the good use of channels. For example, we want to exclude processes like

$$\mathtt{P} \overset{\text{def}}{=} \mathtt{x(y).\overline{y}\langle uv\rangle} \qquad \mathtt{Q} \overset{\text{def}}{=} \mathtt{\overline{x}\langle y'\rangle.y'(w)} \qquad \mathtt{P\,|\,Q}$$

where Q passes a channel to P that Q uses as input of a single name while P tries to send two names along y.

To reason about this type of errors, Milner defined a simple type system for π-calculus in [28] by imposing a *name discipline* through *sorts* and *sortings*. A *sort* is a collection of names that a channel can pass/receive while a *sorting* is a mapping between names and *sorts*.

For the previous example we can say channel x must have sort CHAN(CHAN(INT)) and y sort CHAN(INT) (forming the sorting {x ↦ CHAN(CHAN(INT)), y ↦ CHAN(INT)}) which will make P be ill-typed since it tries to send a pair of values. Thus Milner's *sortings* guarantee a channel is always used correctly w.r.t. its arity and declared type (we can, and will from now on, regard sorts as types). However, taking the HTTP server as example again, it can not prevent the following error:

$$\begin{aligned}&\mathtt{HTTPServer} \overset{\text{def}}{=} \mathtt{handshake().(\nu ch)\overline{init}<ch>.reply(ch)}\\&\mathtt{Thread} \overset{\text{def}}{=} \mathtt{init(x).x(cmd)}\\&\mathtt{Client} \overset{\text{def}}{=} \mathtt{\overline{handshake}<>.reply(x).\overline{x}<GET>}\\&\mathtt{System} \overset{\text{def}}{=} \mathtt{!(HTTPServer\ |\ Thread)\ |\ Client}\end{aligned}$$

where the server instead of replying to the client the channel with the server's thread, it awaits for something along the reply channel thus deadlocking the system.

To be able to forbid these kind of ill-programs we need to add information to channel types that tells us which type of channel we're expecting.

We then have type $[\tau_1, \ldots, \tau_n]\text{chan}_M$ that states a channel for the modality M (input, output or both) is expected with the capability of sending/receiving n values of certain types. A possible syntax of types for the π-calculus is defined in Figure 2.7 and some key typing rules are presented in Figure 2.8. The type ok is used when a program is well-typed. For example, in rule (VAL PAR) we're stating that if program P and program Q are, independently, well-typed then we can compose the two together, P|Q, and obtain a well-typed program. Rules (VAL IN) and (VAL OUT) guarantee a channel has the type we're expecting.

$$\begin{array}{lcl}\sigma & ::= & \tau \mid \text{ok}\\ \tau & ::= & [\tau_1, \ldots, \tau_n]\text{chan}_M \mid \nu \qquad \nu \in \{\text{Int}, \text{Bool}, \ldots\}\\ \text{M} & ::= & ! \mid ? \mid ?!\end{array}$$

Figure 2.7: Syntax of types for π-calculus

$$\frac{\Gamma,\, n : \tau \vdash P \qquad \tau \text{ is a channel type}}{\Gamma \vdash (\nu n)P}\ (\textsc{val new}) \qquad \frac{\Gamma \vdash P : \text{ok} \qquad \Gamma \vdash Q : \text{ok}}{\Gamma \vdash P|Q : \text{ok}}\ (\textsc{val par})$$

$$\frac{\Gamma \vdash \alpha : [\tau_1,\, \dots,\, \tau_n]\text{chan}_! \qquad \Gamma \vdash v_1 : \tau_1 \,\dots\, \Gamma \vdash v_n : \tau_n}{\Gamma \vdash \overline{\alpha}{<}v_1,\, \dots,\, v_n{>} : \text{ok}}\ (\textsc{val out})$$

$$\frac{\Gamma \vdash \alpha : [\tau_1,\, \dots,\, \tau_n]\text{chan}_? \qquad \Gamma,\, x_1 : \tau_1 \,\dots\, x_n : \tau_n \vdash P : \text{ok}}{\Gamma \vdash \alpha(x_1 : \tau_1,\, \dots,\, x_n : \tau_n).P : \text{ok}}\ (\textsc{val in})$$

Figure 2.8: π-calculus' typing rules

In [21], Kobayashi presents a tutorial about the results of channel usage type theory for π-calculus. An important result is the linearity study for π-calculus done by Kobayashi, Pierce and Turner in [22], where channels types are associated with a *multiplicity* to represent the number of times a channel should be used (once or unlimited times).

Although these type systems are rich enough to reason about channel communication individually, they are not sufficient for SOC models where we have structured and context-based communication. It is however a starting point for the kind of type systems we'll need for SOC when regarding channel usage.

2.3.3 Session Types

An initial work for these type systems was developed in [16], later refined in [17], where session types are introduced as the description of the behaviour that takes place in one of the session's endpoints (recall their calculus is based on *binary sessions*). This was achieved by describing the behaviour by means of the participant capability to communicate (channel types) along with a linearity usage on channels based on [22], thus enjoying a *unique handling property* (as presented in [28]).

The syntax of types for the presented fragment (Figure 2.3) is defined in Figure 2.9. A first observation goes to the type $< \alpha >$ that represents the behaviour of one party of a session, namely the party that accepts requests through primitive `accept`. It is enough to know the type

$$\begin{array}{lll}
\upsilon & ::= & {<}\alpha{>} \mid \nu \qquad \nu \in \{\text{Int, Bool}, \dots\} \\
\alpha, \beta & ::= & \downarrow[\overline{\upsilon}];\alpha \mid \downarrow[\alpha];\beta \mid \&\{\text{l}_1\!: \alpha_1, \dots, \text{l}_n\!: \alpha_n\} \mid - \\
 & & \mid \uparrow[\overline{\upsilon}];\alpha \mid \uparrow[\alpha];\beta \mid \oplus\{\text{l}_1\!: \alpha_1, \dots, \text{l}_n\!: \alpha_n\}
\end{array}$$

Figure 2.9: Syntax of types for session π-calculus

of one of the parties since their behaviour is symmetric, *i.e.* the type of one is the complementary type of the other. We then define $\overline{\alpha}$ as the complement type of α as follows:

$$\begin{array}{lll}
\overline{\uparrow [\tilde{v}];\alpha} = \downarrow [\tilde{v}];\overline{\alpha} & \overline{\oplus\{l_i : \alpha_i\}} = \&\{l_i : \overline{\alpha}\} & \overline{\uparrow [\alpha];\beta} = \downarrow [\alpha];\overline{\beta} \\
\overline{\downarrow [\tilde{v}];\alpha} = \uparrow [\tilde{v}];\overline{\alpha} & \overline{\&\{l_i : \alpha_i\}} = \oplus\{l_i : \overline{\alpha}\} & \overline{\downarrow [\alpha];\beta} = \uparrow [\alpha];\overline{\beta}
\end{array}$$

Types defined by α give the expected behaviour of an interaction via a channel. The type - indicates that no further connection is possible at a given name. A type associated with a session channel is a *linear type* meaning the session channel and its complement can only occur in exactly one thread of a parallel composition. For example:

$\overline{\mathrm{k}}![\mathrm{v}].\mathrm{k}?[\mathrm{x}] \mid \mathrm{k}?[\mathrm{x}].\overline{\mathrm{k}}![\mathrm{u}]$

$\overline{\mathrm{k}}![\mathrm{v}].\mathrm{k}?[\mathrm{x}] \mid \mathrm{k}?[\mathrm{x}].\overline{\mathrm{k}}![\mathrm{u}] \mid \overline{\mathrm{k}}![\mathrm{u}].\mathrm{k}?[\mathrm{x}]$

In the first example, channel k is used only once, at a given time, while in the second example channel k is used twice to output a value thus only the first example would be correct according to the linear type of k.

Type α is compatible with type β ($\alpha \asymp \beta$) if each common session channel k is associated with complementary behaviour. This assures all communication done via k is error-free. Types can also be composed by a $\circ$ operator that ensures compatibility by unifying two typing environments together when their types are compatible ($\asymp$). When composed, the type for k becomes $-$ refraining further connection at k. This *type algebra* via $\asymp$ and $\circ$ guarantees linearised usage of channels. We present in Figure 2.10 some typing rules of the type system for sessions.

Notice the typing judgments are slightly different from the previous type systems. Here a type judgment is of the form $\Gamma \vdash \mathrm{P} : \Delta$, that states a term P has typing Δ under the typing environment Γ. This differs from what we've seen so far by its typing Δ that is, in fact, a collection of types and not a type *per se*. The purpose of Δ is to keep track of all the free session channels with their associated type, much like a typing environment but only for session channels. Thus operation $\cup$ extends the typing Δ with a new association.

(VAL ACC) and (VAL REQ) certify each party of the session gets the respective type, *i.e.* the requesting party gets type $\overline{\alpha}$ and the accepting party gets it's complementary type thus ensuring the interaction within the session has no ill behaviour like deadlock.

The typing rules concerning communication are similar to the ones we saw in π-calculus, they guarantee the correct number of parameters, the correct types of each and the correct modality for *each* usage of k instead for *all* usage of the channel. Meaning a session channel k can first output values of type α and later on output values of type (α, β) and still be well-typed under this type system (but *ill-sorted* according to Milner's *sortings* [28]).

(VAL BR) states that if every branch of an interface offered at channel k is well-typed then we can type the interface with a branching type that associates for each label the respective type. (VAL SEL) ensures that the type of P corresponds to the type of the chosen branch. Finally, the

$$\frac{\Gamma \vdash P : \Delta \cup \{k : \alpha\}}{\Gamma,\ \mathtt{a} :< \alpha > \vdash \text{accept a(k) in } P : \Delta} \qquad (\textsc{val acc})$$

$$\frac{\Gamma \vdash P : \Delta \cup \{k : \overline{\alpha}\}}{\Gamma,\ \mathtt{a} :< \alpha > \vdash \text{request a(k) in } P : \Delta} \qquad (\textsc{val req})$$

$$\frac{\Gamma \vdash \overline{e} : \overline{v} \qquad \Gamma \vdash P : \Delta \cup \{k : \overline{\alpha}\}}{\Gamma \vdash k![\overline{e}];P : \Delta \cup \{k :\uparrow [\overline{v}];\alpha\}} \qquad (\textsc{val send})$$

$$\frac{\Gamma,\ \overline{x} : \overline{v} \vdash P : \Delta \cup \{k : \overline{\alpha}\}}{\Gamma \vdash k?[\overline{x}] \text{ in } P : \Delta \cup \{k :\downarrow [\overline{v}];\alpha\}} \qquad (\textsc{val rcv})$$

$$\frac{\Gamma \vdash P_1 : \Delta \cup \{k : \alpha_1\} \ \ldots \ \Gamma \vdash P_n : \Delta \cup \{k : \alpha_n\}}{\Gamma \vdash k \triangleright \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\} : \Delta \cup \{k : \&\{l_1 : \alpha_1,\ \ldots,\ l_n : \alpha_n\}} \qquad (\textsc{val br})$$

$$\frac{\Gamma \vdash P : \Delta \cup \{k : \alpha_j\} \qquad 1 \geq j \leq n}{\Gamma \vdash k \triangleleft l_j;P : \Delta \cup \{k : \oplus\{l_1 : \alpha_1,\ \ldots,\ l_n : \alpha_n\}} \qquad (\textsc{val sel})$$

$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Delta'}{\Gamma \vdash P|Q : \Delta \circ \Delta'} \qquad (\textsc{val par})$$

Figure 2.10: Typing rules for session π-calculus

(VAL PAR) ensures the correct composition of processes in the sense that they won't interfere with each others interactions.

We go back to our example and show its type according to the type system presented. If P has type β_1 and Q has type β_2 then the process defined as `Terminal` has type α while process defined as `User` has its complementary type. Thus the system composed by the two processes has type `console:`$<\alpha>$

α ::= $\downarrow$ [*string string*];&{*dir* :$\uparrow$ [*string*]; *cp* : $\overline{\beta_1}$; *rm* : $\overline{\beta_2}$}
$\overline{\alpha}$ ::= $\uparrow$ [*string string*];$\oplus${*dir* :$\downarrow$ [*string*]; *cp* : β_1; *rm* : β_2}

2.3.4 Types for Multiparty Conversations

A conversation (as the ones we saw in multiparty sessions' calculi and Conversation Calculus) differs from a binary session by the number of participants that can interact through it. Just like there was a need to develop a calculus for these kind of interactions typical of a SOC system, a proper type theory was also required.

In [18] the authors use the *global description of interaction* (here used as global types) and *end-point projection* (here used as local types) notions introduced in previous work [6] by the same authors to reason about the behaviour of a multiparty session where a specific type, the global type, describes the behaviour of the whole conversation and a local type describes the local behaviour of each participant. This was necessary since now sessions have more than two participants thus their type wouldn't describe the whole conversation behaviour. Global types have the form p $\rightarrow$ p': k $\langle$ U $\rangle$.G' where p and p' are participants' identities, $\rightarrow$ the direction of communication, k the channel used, U the type of the message transmitted, and G' the interaction that takes place after. Local types have the same syntax of types as the one presented for session types in [17] save from some adaptations: a session type now keeps the identity of the channel used; and local types T are associated with a participant p through a *located type* (T@p). In their work the communication is asynchronous, this along with multiple participants has to be dealt with in the type discipline to ensure the linearity analysis on channels. To do so the authors present a causality analysis on channels through a *prefix ordering* on global types that imposes sequencing by the prefix ordering only on the actions of the same participant.

Regarding the Conversation Calculus' type system, it makes use of theory from behavioural types (description of resource usage), session types [18] (description of participants behaviour by their capability to communicate) and linear types (channel usage discipline). A merge operation is defined to compose two behavioural types (that describe the interactions within a conversation). This operator is inspired by [18] but takes a whole different approach by not making a distinction between local and global types (there's only conversation types *located* in conversations, *i.e.* associated with a conversation name) as well as by not mapping a type with the identity of the associated participant. This enables a more generic description of a conversation where whoever fulfills the protocol described can be safely composed with other participants' types.

We present the syntax of types for Conversation Calculus in Figure 2.11. A conversation type defines the intended local conversations by means of a behavioural type (fragment B of the syntax of types). Behavioural types share some common types with those of session's like the branching type, select type, inaction type and recursive type. However Conversation Calculus' behavioural types incorporate message types (fragment M) that describe the messages exchanged within a conversation or to the caller conversation. Message types are defined by a polarity p, a label l, a direction d, and the type C of what is communicated. The dual type of a behavioural type B, written $\overline{\text{B}}$, is obtained by swapping polarities in B.

The set of labels used in the communication primitives is split into shared labels, $\mathcal{L}_*$, (used in recursive processes) and plain (or linear) labels, $\mathcal{L}_p$.

A typing judgment has the form P $\vdash$ L | B, where L is a located type that associates conversation types to conversation names (much like typing Δ in session types did) while B is the a behavioural type describing the behaviour of the current context, stating P *expects* type L | B from the *environment*. This last statement differs from saying P *has* type L | B, here we describe what P needs from the environment to be able to behave correctly.

For instance, suppose we have a conversation n where we send a message through label l, n $\blacktriangleleft$ [!l(P)], then if P has type C its type would be n: [?l(C)], stating it expects someone able to receive a message of type C through label l.

$$
\begin{array}{lcl}
\mathrm{B} & ::= & \mathrm{B}_1 \mid \mathrm{B}_2 \ \mid\ \mathbf{0} \ \mid\ \mathrm{rec}\ \mathrm{X.B} \ \mid\ \mathrm{X} \ \mid\ \oplus_{i\in I}\{\ \mathrm{M}_i.\mathrm{B}_i\} \ \mid\ \&_{i\in I}\{\ \mathrm{M}_i.\mathrm{B}_i\} \\
\mathrm{M} & ::= & \mathrm{pl}^{d}(\mathrm{C}) \\
\mathrm{p} & ::= & ! \ \mid\ ? \ \mid\ \tau \\
\mathrm{d} & ::= & \uparrow \ \mid\ \downarrow \\
\mathrm{C} & ::= & [\mathrm{B}] \\
\mathrm{L} & ::= & \mathrm{n}\colon \mathrm{C} \ \mid\ \mathrm{L}_1 \mid \mathrm{L}_2 \ \mid\ \mathbf{0}
\end{array}
$$

Figure 2.11: Conversation Calculus's syntax of types

$$\frac{\mathrm{P} \vdash \mathrm{T}_1 \qquad \mathrm{Q} \vdash \mathrm{T}_2}{\mathrm{P}|\mathrm{Q} \vdash \mathrm{T}_1 \bowtie \mathrm{T}_2} \quad (\textsc{val par})$$

$$\frac{\mathrm{P} \vdash \mathrm{L} \mid \mathrm{B}}{\mathrm{l}^{\mathrm{d}}!(\mathrm{n}).\mathrm{P} \vdash (\mathrm{L} \bowtie \mathrm{n} :\ \mathrm{C}) \mid ?\mathrm{l}^{\mathrm{d}}(\mathrm{C}).\mathrm{B}} \quad (\textsc{val send})$$

$$\frac{\mathrm{P}_{\mathrm{i}} \vdash \mathrm{L} \mid \mathrm{B}_{\mathrm{i}} \mid \mathrm{x}_{\mathrm{i}} :\ \mathrm{C}_{\mathrm{i}}\ (\mathrm{x}_{\mathrm{i}} \notin \mathrm{dom}(\mathrm{L}))}{\Sigma_{\mathrm{i}\in\mathrm{I}}\ \mathrm{l}_{\mathrm{i}}^{\mathrm{d}}?(\mathrm{x}_{\mathrm{i}}).\mathrm{P}_{\mathrm{i}} \vdash \mathrm{L} \mid \oplus_{\mathrm{i}\in\mathrm{I}}\{!\mathrm{l}_{\mathrm{i}}^{\mathrm{d}}(\mathrm{C}_{\mathrm{i}}).\mathrm{B}_{\mathrm{i}}\}} \quad (\textsc{val rcv})$$

$$\frac{\mathrm{P} \vdash \mathrm{L} \mid \mathrm{B}}{\mathrm{n} \blacktriangleleft [\mathrm{P}] \vdash (\mathrm{L} \bowtie \mathrm{n} :\ [\downarrow \mathrm{B}]) \mid \mathrm{loc}(\uparrow \mathrm{B})} \quad (\textsc{val conv})$$

$$\frac{\mathrm{P} \vdash \mathrm{L} \mid \mathrm{B}_1 \mid \mathrm{x} :\ [\mathrm{B}_2]\ (\mathrm{x} \notin \mathrm{dom}(\mathrm{L}))}{\mathrm{this}(\mathrm{x}).\mathrm{P} \vdash \mathrm{L} \mid (\mathrm{B}_1 \bowtie \mathrm{B}_2)} \quad (\textsc{val this})$$

Figure 2.12: Conversation Calculus's typing rules

The merge relation is defined on types to ensure the merging of two types, $\mathrm{T} = \mathrm{T}_1 \bowtie \mathrm{T}_2$, is the behavioural composition of each. The resulting type is usually not unique. This enables us to define service usage protocols using conversation types and allow any service that can fulfill any part of the protocol to execute safely. Also it ensures that both T_1 and T_2 don't interfere with each other when composed together.

We now present the some of typing rules for Conversation Calculus in Figure 2.12.

The rule (VAL CONV) types a piece of conversation, it states process *P* expects some behaviour located in conversations L, and some behaviour in the current conversation. A piece of conversation is then typed with the merging of the located type L with a type that describes the behaviour of the new conversation piece, in parallel with the type of the now current conversation. So basically process P is now localised in conversation n meaning we'll collect a new conversation type for this new conversation. Furthermore, the type of the conversation is obtained by projecting the local behaviour of process P's current conversation type B ($\downarrow$B) and the new current conversation behaviour is described by the remainder of that projection ($\uparrow$ B) after swapping all message type's directions to the current level $\downarrow$.

Essentially rule (VAL RCV) means that since we're offering an interface of possible behaviours defined by B_i, then we expect the environment to choose one of those behaviours, this is translated with a choice type $\oplus$. Also, since it's a guarded summation, we expect the environment to pick one of those behaviours through an output communication hence the prefixing with message types $!l_i^d(C_i)$.

In rule (VAL SEND) we expect the environment to await for a message on label l then proceed with behavioural type B. The located type is a separate view of itself, this is essential to state that we can output part of the type of some conversation, thus enabling partial delegation. Rule (VAL THIS) states that conversation x is a separate view of the current conversation enabling the current conversation to be bound to x.

This type system guarantees that there is no message race error on linear labels, meaning there is always only one possible synchronisation on conversation's communication primitives (linearised usage on channels). This type of analysis in not achieved in Bruni's work [3] but is studied and realised in multiparty session π-calculus [18] through causality analysis on communication channels.

A progress property is also valid for this type system which means a well-typed process in Conversation Calculus never deadlocks.

Thus Conversation Calculus has the dynamic join of participants obtained from a partial delegation semantics as well as the conversation initiation through endpoints that [3] also enjoys off; and a type theory based on [18] that is novel in the sense that the treatment of global and local types is uniform in the merge operation and no participant identity is kept.

Going back to our toy example in Chapter 1 (Figures 1.2 and 1.3), we would obtain the following conversation types for each site (before composing them together):

```
ClientCode ⊢
  WeatherSite: ?forecastWeather([!location().?report()])

WeatherSiteCode ⊢
  WeatherSite: !forecastWeather([!location().τgetReport().?report()])
  | WeatherStation: ?weatherReport([!getReport().?report()])

WeatherStationCode ⊢
  WeatherStation: !weatherReport([!getReport().?report()])
```

The type for `ClientCode` is obtained by using rules (VAL CONV), (VAL PAR) and (VAL SEND) to type the invocation itself and rules (VAL SEND) and (VAL RCV) to get the type of the service's invocation body. For the `WeatherStationCode` we use rules (VAL RCV) and (VAL CONV) to get the service's definition type and obtain its body's type the same way as `ClientCode`. Finally, the type for the `WeatherSiteCode` is a collection of two conversation types, one describes the type of the service definition that we get through the same rules as the `WeatherStationCode` for the declaration part and with rules (VAL THIS), (VAL PAR), (VAL CONV) and (VAL SEND) for the delegation part of its body; and the other defines a type that is required for the join of service `weatherReport` to conversation `WeatherSite`.

After composing these three sites together through the parallel operator we obtain the type:

```
ClientCode | WeatherSiteCode | WeatherStationCode ⊢
     WeatherSite:  τforecastWeather([!location().τgetReport().?report()])
   | WeatherStation:  τweatherReport([!getReport().?report()])
```

This type was obtained through rule (VAL PAR), that composes each site's conversation types using the merge operator ⋈:

```
WeatherSite: ?forecastWeather([!location().?report()])
                                            ⋈
WeatherSite: !forecastWeather([!location().τgetReport().?report()])
| WeatherStation: ?weatherReport([!getReport().?report()])
                                            ⋈
WeatherStation: !weatherReport([!getReport().?report()])
```

and states the invocation of both `forecastWeather` and `weatherReport` as internal actions of the system and the expected behaviour of each service: service `forecastWea-`
`ther` awaits on `location` then an internal action through label `getReport` occurs and, finally, outputs via `report`; and service `weatherReport` inputs on label `getReport` and outputs through `report`.

2.3.5 Type Checking and Type Inference

Type systems *per se* are useless in practice if there isn't a method to verify, at compile time or run-time, whether or not a program is well-typed.

A *typing algorithm* is then used to check whether a given program is well typed (*i.e.* obeys all typing rules of the type system), or not. To aid the algorithm and reduce its complexity, programs can be explicitly annotated with types but, on the other hand, the syntax can also become quite cumbersome with all the annotations. This has motivated the study of *type reconstruction algorithms* which given a program with less to none annotations, infers the types necessary to type check.

To be able to determine types, in this context, *type variables* are used as placeholders for the omitted types and *type substitutions* σ map types to type variables, thus the idea of type inference consists in finding valid instantiations (*i.e.* that makes the program well typed) to all type variables.

A typical algorithm for type inference is *Algorithm W* due to Milner [26], later proven complete and extended by Milner and Damas [9]. Algorithm W separates the problem of type inference into two subproblems: the generation of equations between types variables and types (*constraints*), and finding a solution to the equations.

The former is known as *constraint-based typing* where a set of *constraint typing rules* are used to generate constraints. A solution is a pair of a type substitution σ and type T such that applying the substitution to S gives us type T, where S is the type of term containing type variables (together with the type environment Γ, forms the input of the algorithm). To find a solution to a set of constraints on types, an *unification algorithm* [35, 12, 24, 8] is used that always returns the most general solution (*principal unifier*) from which any other solution can be obtained. Then we always obtain the most general type (*principal type*) T by applying the *principal unifier* to S. Usually, the generation and attempt to solve the constraints is done interleaved so as to return a principal type at each step.

Most modern languages use a *polymorphism* mechanism in order to use code with different types in different contexts. In particular, *let-polymorphism* which is a form of *parametric polymorphism* (*i.e.* using type variables to be able to instantiate with the types needed at any given context) that allows for polymorphic let-binding values.

The extension of Algorithm W by Milner and Damas already treats *let-polymorphism*.

Another kind of polymorphism is *subtyping polymorphism*, or just *subtyping*, which is also used in most modern languages to define a relation between types that states that if we expect type T and know T is a subtype of U then we can safely use U where T was expected. This is known as the *subsumption* rule in a type system with subtyping. On the other hand, the subtyping relation is defined by its own inference rules (usually one or more for each kind of type) enjoying of both reflexive and transitive properties.

These last two properties of the subtyping relation together with the subsumption rule make the typing rules and subtyping rules not *syntax directed* thus unsuitable for implementation due to the non-determinism when applying these rules. To solve this issue, we must apply *algorithmic* typing and subtyping as opposed to their *declarative* counterparts. In *algorithmic* subtyping, all the rules concerning a specific type are put together in a single rule whilst in *algorithmic* typing we eliminate the subsumption rule to incorporate it directly in the typing rules where we can apply subsumption. So, in general, *subtyping* doesn't increase the *typechecking* complexity.

However, type inference with subtyping have been proven hard especially if we want to ensure a principal type as a result. Mishra and Fuh described the issue in [11] through a simple example: if t = λ x.x then a type inference algorithm will give us the principal type $\tau = \alpha \to \alpha$. One type t has is int → real since int <: real however this is not a substitution of the principal type.

To deal with this issue, the type inference algorithm must have constrained quantification so that quantified type variables can only have instantiations that respect a set of constraints. This technique allows us to find a principal type when dealing with subtyping but makes the inferred type more verbose since it will include a constraint to define which kind of instantiations are possible.

Smith in [37] studied an extension of Algorithm W to include subtyping along with some techniques to simplify the resulting type. Sulzmann, Odersky and Wehr in [31] presented a general framework, HM(X), that defines a family of type systems based on Algorithm W, this framework is instantiated with a specific constraint system X. The authors define the conditions necessary on X to ensure HM(X) type system preserves the original properties of always finding a principal type as well as to obtain an inference algorithm. In particular, they studied a case with subtyping and prove it retains these enjoyable properties. In [34], Pottier introduces some simplifications to the obtained types by eliminating some of the (unnecessary) constraints.

In [15] (a revision and correction of previous work in [14]) a study regarding subtyping of session types for a π-calculus extended with session types was presented. The authors used the notions of channel subtyping from [33], furthermore they also define a typechecking and type inference algorithm for their extended π-calculus. In [13] the author further increases the expressiveness of the extended π-calculus with session types [15] by introducing the notion of polymorphism in session types. This allows for generic service's usage protocols that can be instantiate with a type when a client makes an invocation.

Another preliminary study on type inference for session types was presented by Mezzina in [25] based on SCC calculus. Because duality of sessions types, branching and selection (external and internal choice) are not directly related to the unification, the author introduces a new kind of constraint on types and proposes an algorithm to solve these.

More recently, Mostrous, Yoshida and Honda presented in [29] an algorithm for the subtyping relation of multiparty sessions [18] as well as a type inference algorithm that determines the principal global description from end-point processes which is minimal w.r.t. subtyping.

Finally, some proposals and implementations for typed languages based on session types were made in the last years. In [10] the authors propose an object-oriented language that integrates session types in their type system. This accommodation is obtained by extending class and methods signatures to include channel usage protocol described by session types. The language lacks common object-oriented features like exceptions, polymorphism and recursive types. The authors in [39, 40] propose a multi-threaded functional language that incorporates session types while the authors of [30] take a different approach by introducing session types in Haskell by means of encoding a session type to a Haskell type, therefore making use of its typechecker to ensure type safety. An extension to Java to integrate session types was implemented and presented in [19], this works follows from [17]. Up until now, there was not an implementation for multiparty conversations. This dissertation's work tackles this issue with the implementation of a prototype typed distributed programming language for conversation types.

3 . Distributed Conversations: A Programming Language for Multiparty Conversations

In this chapter we overview the design of the proposed prototype for a programming language with distributed runtime support for multiparty conversations. We will start by introducing the language's syntax and, informally, its semantics. We conclude with some examples to illustrate some of the language's expressiveness.

3.1 Language Design

Our language is focused, broadly speaking, in services that can be defined (*published*), invoked (*discovered*) and delegated by means of a *conversation* abstraction. Consequently, every communication has a context thus we have structured context-sensitive interactions amongst multiple participants (this is the very concept of *conversation*). This kind of structured communication is found in services' interactions rather than client-to-client communications, therefore we naturally adopt it in our language design as did the authors of the process calculi (CC) on which this language is based on.

3.1.1 Syntax and Semantics

A program unit in our language consists in RemoteTypes; Sites; Statement;; where RemoteTypes are declarations of remote services' types, Sites are declarations of domain spaces (*sites*) where local services can be declared, and Statement is the client's code.

In Figure 3.1 we show the syntactical category of our language's grammar that corresponds to statements, Figure 3.3 shows the expressions of our language, Figure 3.4 the values our language has, and finally Figure 3.5 are the types we can declare as remote types. We will now explain the semantics, rather informally, of some key constructs of our language.

We begin by looking at the *site definition* construction

site #n { Statement }

This declares a *namespace* for service's definition. It is identified by a *label* which is a special string (a string started with a #) that we use to name services, sites and communication labels. The *service definition* primitive

def #s **as** { Statement }

can only occur inside a site definition. Its purpose is to define a service that can be invoked both locally or remotely, and when so it runs the code in Statement on the site where it was defined. Furthermore the code is only evaluated when the service is invoked. The service has a name associated that only needs to be unique w.r.t. a site, *i.e.* different sites can have the services that share the same name (even if they're all located in the same machine).

```
Program        ::=  RemoteTypes; Sites; Statement;;
RemoteTypes    ::=  remoteTypes Types
                  | RemoteTypes; RemoteTypes
Sites          ::=  site <LABEL> { Declarations }
                  | Sites; Sites
Declarations   ::=  Declarations; Declarations
                  | def <LABEL> as { Statement }
Statement      ::=  Statement; Statement
                  | var x = Expression
                  | val x = Expression
                  | Expression = Expression
                  | { Statement } || ... || { Statement }
                  | select {<LABEL>: Statement;
                            <LABEL>: Statement;
                            (<LABEL>: Statement)*
                  | switch { case ( Expression ) do <LABEL>: { Statement };
                                           ...
                             case ( Expression ) do <LABEL>: { Statement };
                             default do <LABEL>: {Statement } }
                  | while(Expression) do { Statement }
                  | if(Expression) then { Statement } else { Statement }
                  | fun <ID>(<ID>, ..., <ID>) = { Statement }
                  | return Expression
                  | Expression(Expression, ..., Expression)
                  | print Expression
                  | println Expression
                  | send(<LABEL>, Expression)
                  | sendUp(<LABEL>, Expression)
                  | invoke <LABEL> in <LABEL> as { Statement }
                  | join <LABEL> in <LABEL> as { Statement }
```

Figure 3.1: Statement's fragment of our language's syntax.

To invoke a service we can use one of two constructions

invoke #s **in** http://ip_address:port/#n **as** { Statement }

or

join #s **in** http://ip_address:port/#n **as** { Statement }

Both need to know the location (ip address and port) of the service they intend to invoke and run the code defined in `Statement`. As we said before, we make use of *conversations* to give context to interactions between peers in the context of service oriented computations. These are hidden to the user and only used by the language's runtime support. Therefore, the difference between the service invocation methods lies in the conversation usage: while the former *creates* a new conversation abstraction, the latter uses the *current* conversation to invoke the intended service (so we can only use this method of invocation inside an on-going conversation). Thus the primitive `join` allows us to *delegate* work to a third-party service by asking this service to join the current conversation.

For communication we have **send**(l, Expression) to send a value through label l in the *current* conversation (*i.e.* only participants of the same conversation can capture this communication); **sendUp**(l, Expression) to send a value through label l in the *enclosing* conversation (for *e.g.* if we start a new conversation within an existing one we can use this construction to communicate with the upper level conversation); **receive**(l) to receive a value through label l in the *current* conversation; **receiveUp**(l) to receive a value through label l from the *enclosing* conversation.

The communication is synchronous and a label can only synchronise with a label that is equal but of different polarity (*i.e.* sending a value through label l can only synchronise via a receive in the same label). Note that it is possible to have different services using the same name for a label because communication is done within a conversation created when the service is invoked, meaning different invocations of the same service or different services using the same label name won't interfere with each other.

We allow users to define an interface of options within their service definitions (or even in a service invocation method) through a select construction

select {
  l_1: { Statement };
  l_2: { Statement };
  (l_i: { Statement };)$*$
}

A select is composed by branches of the form l: { Statement }, where l is a label and Statement the associated code. The code within a branch is only executed if the corresponding label is selected. This can be seen as guarded code that is only executed if their guard is synchronised. To select an option from a select construction we use the switch primitive

```
switch {
   case (Expression) do $l_1$: { Statement };
   ...
   case (Expression) do $l_n$: { Statement };
   default do $l_i$: { Statement }
}
```

The switch construction is composed of branches of the form **case** (Expression)**do** l: { Statement } and one default branch of the form **default do** l: { Statement }. Each expression on a case clause represents a condition to select a specific option of the select construction. This selection means that the code associated with the select option is executed on both the select and switch side (corresponding to a synchronisation of a guard in a guarded code). For instance, on the following snippet (Figure 3.2) we have a service that offers three options and a client of that service that selects one of those options:

```
site #siteA {
   def #serviceA as {
      ...
      select {
         #option$_1$: {
            Code$_1$
         },
         #option$_2$: {
            Code$_2$
         },
         #option$_3$: {
            Code$_3$
         }
         ...
```

(a) Service's side

```
invoke #serviceA in htpp://.../#siteA as {
   ...
   switch {
      case (cond$_1$) do #option$_1$: {
         Code$'_1$
      },
      case (cond$_2$) do #option$_2$: {
         Code$'_2$
      },
      case (cond$_3$) do #option$_3$: {
         Code$'_3$
      },
      default do #option$_2$: {
         Code$_{\text{default}}$
      }
      ...
```

(b) Service's client's side

Figure 3.2

To determine which branch to execute, the case clauses on the client side are evaluated sequentially until one of the conditions is true, otherwise the default branch is selected. Let's assume cond$_1$ and cond$_2$ have the values false and true, respectively. Then the client will choose option$_2$ and execute the respective code Code$'_2$ while on the service's side the code Code$_2$ is executed.

We also have **if**(Expression)**then** { Statement } **else** { Statement } instructions working as a switch, with only two branches and no default branch, when the the **if**'s branches have communication primitives (**send** or **receive**), for *e.g.*:

```
if(Expression) then {
  ...
  send(#l, 0);
  ...                                switch {
} else {                  ⟹            #l : { send(#l); ... };
  ...                                  #l′: { val x = receive(#l′); ... }
  val x = receive(#l′);               }
  ...
}
```

Finally, we have a parallel operator to execute code in parallel

{ Statement } || ... || { Statement }

It is important to note that any variable declared prior to the use of the operator will have a copy inside an executing thread. This is to avoid race conditions and to simplify the implementation of the operator. Otherwise specific methods to synchronise the code run in parallel would have to be implemented.

Our language has constants, booleans, strings and arrays as values (Figure 3.4). To create an array we use the construction

array(Expression_1, Expression_2)

where first expression is the size of the array and the second the initial value of every position in the array.

3.1.2 Distributed Runtime Support

As we stated in section 3.1, communications in our language are aware of their context and structured through a *conversation* abstraction. Furthermore, a *conversation* starts upon a service invocation and exists while there are still participants in that conversation. This abstraction is however hidden from the user and handled by the distributed runtime support of our language. The programmer does not have to concern himself/herself with issues like: when to start a conversation, to whom send a message, from whom receives a message, etc.

So our language's runtime support has the following tasks:

- Initiate a *conversation* by handling a unique (throughout the network) conversation identifier;
- Invoke local services;
- Keep routing information;
- Keep information about local services;
- Transparently communicate with other partners in a conversation;

```
Expression ::= receive(<LABEL>)
             | receiveUp(<LABEL>)
             | if(Expression) then { Expression } else { Expression }
             | Expression(Expression, ..., Expression)
             | Expression and Expression
             | Expression or Expression
             | not Expression
             | Expression < Expression
             | Expression <= Expression
             | Expression == Expression
             | Expression != Expression
             | Expression > Expression
             | Expression >= Expression
             | Expression + Expression
             | Expression − Expression
             | Expression ∗ Expression
             | Expression / Expression
             | Expression % Expression
             | – Expression
             | array( Expression, Expression)
             | Expression[Expression]
             | length( Expression )
             | !Expression
             | ( Expression )
             | <ID>
             | Value
```

Figure 3.3: Expressions's fragment of our language's syntax.

```
Value ::= <CONSTANT>
        | <BOOL>
        | <STRING>
        | Array(Values)
```

Figure 3.4: Values's fragment of our language's syntax.

```
Types             ::= <LABEL>:[<LABEL>](BehaviouralType)
BehaviouralType ::= BehaviouralType; BehaviouralType
                    | BehavouralType|BehavouralType
                    | <LABEL>?(BasicTypes)
                    | <LABEL>!(BasicTypes)
                    | <LABEL>?^(BasicTypes)
                    | <LABEL>!^(BasicTypes)
                    | +{<LABEL>: BehaviouralType, ..., <LABEL>: BehaviouralType}
                    | &{<LABEL>: BehaviouralType, ..., <LABEL>: BehaviouralType}
BasicTypes        ::= Int
                    | Bool
                    | String
                    | Array(Types)
                    | String
                    | _
```

Figure 3.5: Types's fragment of our language's syntax.

- Send routing information to other partners in a conversation;
- Ensure the messages received by a partner are correct w.r.t. their ordering and their context (conversation they belong to)

3.2 Examples

We now present some small examples using our language.

3.2.1 The Calculator Service

In this example we have a calculator service that offers four operations upon invocation: sum, subtraction, multiplication, and division.

```
site #operations {
  def #sum as {
    val op1 = receive(#op1);
    val op2 = receive(#op2);
    send(#res, op1 + op2)
  }
};
```

Each operation is itself a service defined, in a site named “#operations”, as a service that receives two operands and sends back the result of applying the operation to those operands. For simplification we only present one of such services, the sum, and will let the calculator service do the other operations instead.

```
site #utilities {
  def #calculator as {
    val op1 = receive(#l1);
    val op2 = receive(#l2);
    select {
      #sum: {
        join #sum in http://localhost:8000/#operations as {
          send(#op1, op1);
          send(#op2, op2)
        }
      };
      #sub: { send(#res, op1−op2) };
      #mul: { send(#res, op1∗op2) };
      #div: { send(#res, op1/op2) }
    }
  } };;
```

The calculator service is available at a site named "#utilities". Upon invocation, the service will receive two values and offer a menu of options to the client, each option represents one possible operation. In particular, if the client opts to do a sum, the calculator service will invoke the respective service through a join primitive. In other words, the service that defines a sum joins the conversation between the client and the calculator service, extending the conversation to a third participant. This enables the sum operation service to send the result directly to the client.

```
remoteType #utilities:[#calculator](#l1?(Int);#l2?(Int); &{#div: #res!(Int), #mul:
    #res!(Int), #sub: #res!(Int), #sum: #res!(Int)})
invoke #calculator in http://10.170.136.73:8000/#utilities as {
  send(#l1, 3);
  send(#l2, 5);
  var r = 0;
  switch{
    case(false) do #mul: { r = receive(#res) };
    case(true) do #sum: { r = receive(#res) };
    case(false) do #div: { r = receive(#res) };
    case(false) do #sub: { r = receive(#res) };
    default do #mul: { r = receive(#res) }
  };
  println !r
};;
```

The client invokes the calculator service located at some IP address (the client needs to know beforehand this address) and send the values he wishes to compute. Then he chooses one of the offered operations through a criteria, in this case the client just wants to do a sum so all other option's criteria is a boolean false. Due to lack of subtyping and to be able to typecheck our programs, we require that the client always take into consideration all the offered options even if it's only interested in a subset of said options. This means that each switch primitive needs to have at least the same distinct options as its counterpart select primitive. Finally, after picking the desired operation, the client receives the resulting value, unaware that it is receiving it from a third participant.

3.2.2 The Online Support Service

For this example we have an ISP company that provides an online service, #onlineService, that can either provide technical support or give commercial information regarding their products. In particular, the technical support is provided by a "specialist" obtained through a remote service, #Assistant.

```
remoteType #Support:[#Assistant](#product?(_);#askForProblem!(String);
    #problemDetails?(_);#solution!(String);#done!(String))

site #CompanyHQ {
  def #onlineService as {
    val clientInfo = receive(#info);
    select{
      #support: {
        val v = receive(#support);
        join #Assistant in http://localhost:8000/#Support as {
          val ok = receive(#done);
          send(#bye, "Thanks for using our online support service.")
        }
      };
      #commercial: {
        val v = receive(#commercial);
        val product = receive(#product);
        print "Getting details for product: "; println product;
        send(#details, "Details about product...");
        send(#bye, "Thanks for using our online support service.")
      }
    }
  }
};;
```

Should technical support be requested, the main service will ask the `#Assistant` service to join the conversation between the client and the `onlineService`. At this point, communication is done between the new participant, the technical support assistant, and the client (who is unaware of a third participant) until a solution is provided, then the main service courteously terminates the service and, since no more interactions happen between the participants, the conversation.

```
site #Support {
  def #Assistant as {
    val product = receive(#product);
    print "Getting help request for product: "; println product;
    send(#askForProblem, "How can I help you?");
    val problemDetails = receive(#problemDetails);
    send(#solution, "reset modem");
    send(#done, " ")
  }
};;
```

The technical support assistant service, upon invocation, receives the product's information and enquires the client about its problem, providing afterwards the typical "solution" to the problem.

```
fun supportDepartment(help, product) = {
  if(help == true) then {
    send(#support, " ");
    send(#product, product);
    val enquire = receive(#askForProblem);
    send(#problemDetails, "Modem has no signal.");
    val solution = receive(#solution);
    val end = receive(#bye);
  }else {
    send(#commercial, " ");
    send(#product, product);
    val details = receive(#details);
    val end = receive(#bye)
  }
};

{
  invoke #onlineService in http://localhost:8000/#CompanyHQ as {
    send(#info, "N01");
    supportDepartment(true, 0123456789) }
}
||
{
  invoke #onlineService in http://localhost:8000/#CompanyHQ as {
    send(#info, "N02");
    supportDepartment(false, "router") }
};;
```

Finally, we have two clients on the same machine running in parallel. Each client invocation of service `onlineService` will create a fresh conversation with the service. So all interactions between a client and the service are located in the conversation generated and can not be captured by the other client. We encapsulate most of the client's code in a function named `supportDepartment` where depending on the `help` argument, the client chooses either to request support or to request commercial information. Notice that the `if` instruction, in this case, works as a switch with only two branches and no default branch and that the function is polymorphic and therefore it can be used with it's `product` parameter as a string (product's name) or an integer (product's id).

3.3 Known Limitations

Some limitations to the language's expressiveness are known, such as:

1. An if-then-else with behaviour corresponds to an internal choice but this is actually not supported by our runtime system;
2. An array of strings that contain commas (",") is not correctly unmarshalled in our runtime system;
3. We can not typecheck service definitions that invoke services defined under the same site.

The first issue is due to the fact that during the implementation phase of the runtime support, we did not foresee that an if-then-else could be seen as an internal choice (*i.e.* switch primitive) so we did not take all the steps necessary to its implementation. The second is because of how we do the marshalling of array values: we create a string with the elements of the array separated by commas. While this could be easily overcome by changing the delimiter of our marshalled arrays, since a string could also contain any other possible delimiter, we found the issue would not be truly overcome by just changing delimiters. However, these limitations persist only due to the lack of time to properly deal with them when taking into consideration that the main focus of this work is in the typechecking algorithm.

4 . Type System and Typechecking for Conversation Types

In this chapter we overview the design of the typechecking algorithm. We start by defining our type system followed by the typechecking and type inference algorithms. We also show some interesting results of the algorithms proposed, namely we prove their soundness and completeness w.r.t. the theory.

4.1 Type System

Our type system is based on the type system of CC's, [4]. Our syntax of types is showed in Figure 4.1. The main difference from CC's consists in the introduction of the sequential composition of types. This is necessary in order to obtain a deterministic unification algorithm. A conversation type, n:[s](B), consists in an association between a site's name n and the behavioural type B for each service s defined in it.

We are interested in using our type system to prevent error programs as those defined in Definition 4.1.3: programs where, at any instant, labels are not used linearly.

Definition 4.1.1 (Initial Message Set). *We define the initial message set on a behavioural type B, denoted as $\mathcal{I}(B)$, as follows:*

$$\begin{array}{lcl}
\mathcal{I}(\emptyset) & ::= & \emptyset \\
\mathcal{I}(lp^d(\beta)) & ::= & lp^d \\
\mathcal{I}(B_1;B_2) & ::= & \mathcal{I}(B_1) \\
\mathcal{I}(B_1|B_2) & ::= & \mathcal{I}(B_1)\cup\mathcal{I}(B_2) \\
\mathcal{I}(\oplus\{l_1:B_1,\dots,l_n:B_n\}) & ::= & \mathcal{I}(B_1)\cup\ldots\cup\mathcal{I}(B_n) \\
\mathcal{I}(\&\{l_1:B_1,\dots,l_n:B_n\}) & ::= & \mathcal{I}(B_1)\cup\ldots\cup\mathcal{I}(B_n)
\end{array}$$

Definition 4.1.2 (Static Context). *Let P be a program, $i,j \in \{1,\dots,n\}$. We then define static program's contexts, $\mathcal{C}[\cdot]$, as:*

$$\begin{array}{lcl}
\mathcal{C}[\cdot] & ::= & \cdot \ \mid \ \mathcal{C}'[\cdot] \\
\mathcal{C}'[\cdot] & ::= & \texttt{def s as}\,\{\,\mathcal{C}[\cdot]\,\} \ \mid \ \texttt{invoke s in n as}\,\{\,\mathcal{C}[\cdot]\,\} \ \mid \ \texttt{join s in n as}\,\{\,\mathcal{C}[\cdot]\,\} \\
 & & \mid \ \mathcal{C}[\cdot];\texttt{P} \ \mid \ \{\,\mathcal{C}[\cdot]\,\}\,\|\,\{\,\texttt{P}\,\} \ \mid \ \texttt{if(E) then}\,\{\,\mathcal{C}[\cdot]\,\}\,\texttt{else}\,\{\,\texttt{P}\,\} \\
 & & \mid \ \texttt{if(E) then}\,\{\,\texttt{P}\,\}\,\texttt{else}\,\{\,\mathcal{C}[\cdot]\,\} \ \mid \ \texttt{fun f(E}_1,\dots,\texttt{E}_\texttt{n}) = \{\,\mathcal{C}[\cdot]\,\} \\
 & & \mid \ \texttt{select\{ l}_1:\texttt{P}_1;\dots,\texttt{l}_\texttt{i}:\mathcal{C}[\cdot];\dots,\texttt{l}_\texttt{n}:\texttt{P}_\texttt{n}\,\} \\
 & & \mid \ \texttt{switch\{ case(E}_1)\ \texttt{do l}_1:\texttt{P}_1;\dots,\texttt{case(E}_\texttt{i})\ \texttt{do l}_\texttt{i}:\mathcal{C}[\cdot];\dots, \\
 & & \qquad \texttt{case(E}_\texttt{n})\ \texttt{do l}_\texttt{n}:\texttt{P}_\texttt{n},\ \texttt{default do l}_\texttt{j}:\texttt{P}_\texttt{j}\,\} \ \mid \ \mathcal{C}''[\cdot] \\
\mathcal{C}''[\cdot] & ::= & \texttt{site n}\,\{\,\mathcal{C}'[\cdot]\,\}
\end{array}$$

Definition 4.1.3 (Error Program). *Let P, Q be a programs of our language, T, T' their respective types, and $\mathcal{C}[\cdot]$ a static context. Then P' is an error program where there are two labels from the union of T's and T''s initial message set such that each share the same direction and polarity.*

$P' = C$[{ P } || { Q }]

$P' = C''$[def s as { P }];C[invoke s in n as { Q }]

$P' = C''$[def s as { P }];C[join s in n as { Q }]

This means that a well-typed program will never have, at any instant, more than a sender or receiver on the same label (within a conversation).

Definition 4.1.4 (Typing Environment)**.** *We define the typing environment, Γ, as the set of associations between identifiers and their types, including conversation types, and a specific conversation type to associate a behaviour to the current context (labelled as this).*

A typing judgment has the form $\Gamma \vdash P : T$ where Γ is a set of type declarations, P is the program to be typed and T is a type. In general Γ contains types for remote services, declared in program P using the **remoteType** primitive, and the specific conversation type $this : B$.

As we stated previously, our type system is an adaptation of the CC's. Namely, ours does not include subtyping and recursive types. Even though we extend the type theory with an additional sequential composition on behavioural types and typing rules for sequential code, this extension is conservative w.r.t. the theoretical results present on [4]. This is because sequential behaviour does not compromise linear usage of labels. Furthermore, type equality is up to permutation of branches in parallel composition of types and internal/external choices, and up to renaming of type variables.

Lemma 4.1.5 (Subject Reduction)**.** *Let P be a program and T a type such that $\Gamma \vdash P : T$. If $P \to Q$ then there is type T' such that $T \to T'$ and $\Gamma \vdash Q : T'$.*

Proof. See proof in [5], Theorem 3.20. ☐

Lemma 4.1.6 (Type Safety)**.** *Let P be a program such that $\Gamma \vdash P : T$ for some T. If there is Q such that $P \to^* Q$, then Q is not an error program.*

$$\begin{array}{lcl}
C & ::= & [s](B) \\
B & ::= & B_1 \mid B_2 \;\mid\; \mathbf{0} \;\mid\; B_1;B_2 \;\mid\; M \\
 & & \oplus\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \;\mid\; \&\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \\
M & ::= & lp^d(\beta) \\
\beta & ::= & Int \;\mid\; Bool \mid String \mid Array(\beta) \mid \beta \xrightarrow{B} \beta' \mid Ref(\beta) \mid Unit \\
p & ::= & ! \;\mid\; ? \;\mid\; \tau \\
d & ::= & \uparrow \;\mid\; \downarrow
\end{array}$$

Figure 4.1: Syntax of Types

Proof. See proof in [5], Corollary 3.24. □

Lemma 4.1.7 (Weakening)**.** *If a program typechecks under a typing environment then it also typechecks under the augmentation of the said typing environment:*

$$\Gamma \vdash P : T \quad \Rightarrow \quad \Gamma \cup \Gamma' \vdash P : T, \textit{for all}\ \Gamma'$$

Proof. Since it is enough to have the typing environment Γ to typecheck a program P then even if we add more mappings, Γ′, to Γ we will have enough information to typecheck a program P. □

Lemma 4.1.8 (Substitution Soundness)**.** *The application of a substitution to a typing judgment does not change its validity:*

$$\Gamma \vdash \mathrm{P} : \mathrm{T} \quad \Rightarrow \quad \theta(\Gamma) \vdash \mathrm{P} : \theta(\mathrm{T})$$

Proof. A substitution only affects a typing judgement if it contains a type variable that occurs in Γ and/or in T, but in those cases what we have is a simple renaming of a type variable to a specific type in all its occurrences and so it does not change the meaning of a typing rule. We will prove for two cases to demonstrate our intuition:

$$\frac{}{\Gamma,\ \textit{this} : l?(\beta) \vdash \texttt{receive(l)} : \beta} \quad (\textsc{recv})$$

Let $\Gamma' = \Gamma,\ \textit{this} : l?(\beta)$, then in this rule if we apply a substitution σ to Γ′ we will either: (1) obtain exactly the same typing environment if $Dom(\sigma) \cap Var(\Gamma') = \emptyset$; (2) or, let $[T/K]$ be a mapping occurring in σ and $\beta = K$ then we would obtain $\Gamma,\ \textit{this} : l?(T) \vdash \texttt{receive(l)} : \texttt{T}$ which is also a valid typing judgment; (3) or, finally, if σ contains a mapping to some type variable occurring in Γ then the typing judgment would also be valid.

$$\frac{\Gamma \vdash E : \beta}{\Gamma,\ \textit{this} : l!(\beta) \vdash \texttt{send(l,E)}} \quad (\textsc{send})$$

Let $\Gamma' = \Gamma,\ \textit{this} : l!(\beta)$, then for this rule we can reason as we did the case before and by applying the substitution either get: (1); or (3); or let $[T/K]$ be a mapping occurring in σ and $\beta = K$ then we would obtain $\Gamma,\ \textit{this} : l!(T) \vdash \texttt{send(l,E)} : \texttt{T}$ with $\Gamma \vdash E : T$ as the premise for the conclusion, which is also a valid typing judgment; □

4.1.1 Typing Rules

We now present the typing rules for our language.

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \qquad \Gamma,\ \mathit{this} : B_2 \vdash P_2}{\Gamma,\ \mathit{this} : B_1;B_2 \vdash \mathtt{P_1;P_2}} \quad (\textsc{seq})$$

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash E : \beta \qquad \Gamma,\ \mathit{this} : B_2,\ x : \beta \vdash P}{\Gamma,\ \mathit{this} : B_1;B_2 \vdash \mathtt{let\ x = E\ in\ \{P\}}} \quad (\textsc{let})$$

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \qquad \ldots \qquad \Gamma,\ \mathit{this} : B_n \vdash P_n}{\Gamma,\ \mathit{this} : B_1 \bowtie \ldots \bowtie B_n \vdash \mathtt{P_1 || \ldots || P_n}} \quad (\textsc{par})$$

$$\frac{\Gamma,\ \mathit{this} : B \vdash P}{\Gamma,\ n : B \vdash \mathtt{site\ n\ \{P\}}} \quad (\textsc{site})$$

$$\frac{\Gamma,\ \mathit{this} : B,\ n : [s](B_1) \vdash P \qquad B_1 = \overline{\downarrow B}}{\Gamma,\ \mathit{this} : \mathit{loc}(\uparrow B),\ n : [s](B_1) \vdash \mathtt{invoke\ s\ in\ n\ as\ \{P\}}} \quad (\textsc{invoke})$$

$$\frac{\Gamma,\ \mathit{this} : B,\ n : [s](B_1) \vdash P}{\Gamma,\ \mathit{this} : B_1 \bowtie B,\ n : [s](B_1) \vdash \mathtt{join\ s\ in\ n\ as\ \{P\}}} \quad (\textsc{join})$$

$$\frac{\Gamma,\ \mathit{this} : B \vdash P}{\Gamma,\ \mathit{this} : [s](\downarrow B);\mathit{loc}(\uparrow B) \vdash \mathtt{def\ s\ as\ \{P\}}} \quad (\textsc{def})$$

$$\frac{}{\Gamma,\ \mathit{this} : l?(\beta) \vdash \mathtt{receive(l)} : \beta} \quad (\textsc{recv})$$

$$\frac{\Gamma \vdash E : \beta}{\Gamma,\ \mathit{this} : l!(\beta) \vdash \mathtt{send(l,E)}} \quad (\textsc{send})$$

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \qquad \ldots \qquad \Gamma,\ \mathit{this} : B_n \vdash P_n}{\Gamma,\ \mathit{this} : \&\{l_1 : B_1;\ \ldots;\ l_n : B_n\} \vdash \texttt{select}\ \{\texttt{l}_1 : \texttt{P}_1;\ \ldots;\ \texttt{l}_\texttt{n} : \texttt{P}_\texttt{n}\}} \quad (\textsc{select})$$

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \mathit{Bool} \qquad \ldots \qquad \Gamma \vdash E_n : \mathit{Bool}\\ \Gamma,\ \mathit{this} : B_1 \vdash P_1 \qquad \ldots \qquad \Gamma,\ \mathit{this} : B_n \vdash P_n\\ \Gamma,\ \mathit{this} : B_d \vdash P_d \qquad \mathit{if}\ l_i = l_j\ \mathit{then}\ B_i = B_j \qquad i, j \in \{1,\ \ldots,\ n\} \qquad m <= n\end{array}}{\begin{array}{c}\Gamma,\ \mathit{this} : \oplus\{l_1 : B_1;\ \ldots;\ l_m : B_m\} \vdash\\ \texttt{switch}\ \{\texttt{case}\ (\texttt{E}_1)\ \texttt{do}\ \texttt{l}_1 : \texttt{P}_1;\ \ldots;\ \texttt{case}\ (\texttt{E}_\texttt{n})\ \texttt{do}\ \texttt{l}_\texttt{n} : \texttt{P}_\texttt{n};\ \texttt{default}\ \texttt{do}\ \texttt{l}_\texttt{i} : \texttt{P}_\texttt{d}\}\end{array}} \quad (\textsc{switch})$$

$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma,\ \mathit{this} : l_1; B_1 \vdash E_1 : \beta \qquad \Gamma,\ \mathit{this} : l_2; B_2 \vdash E_2 : \beta}{\Gamma,\ \mathit{this} : \oplus\{l_1 : l_1; B_1,\ l_2 : l2; B_2\} \vdash \mathit{if}\ (E)\ \mathit{then}\ E_1\ \mathit{else}\ E_2 : \beta} \quad (\textsc{ifE-behaviour})$$

$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma,\ \mathit{this} : l_1; B_1 \vdash P_1 \qquad \Gamma,\ \mathit{this} : l_2; B_2 \vdash P_2}{\Gamma,\ \mathit{this} : \oplus\{l_1 : l_1; B_1, l_2 : l_2; B_2\} \vdash \mathit{if}\ (E)\ \mathit{then}\ P_1\ \mathit{else}\ P_2} \quad (\textsc{if-behaviour})$$

$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash E_1 : \beta \qquad \Gamma \vdash E_2 : \beta}{\Gamma \vdash \mathit{if}\ (E)\ \mathit{then}\ E_1\ \mathit{else}\ E_2 : \beta} \quad (\textsc{if-e})$$

$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash \mathit{if}\ (E)\ \mathit{then}\ P_1\ \mathit{else}\ P_2} \quad (\textsc{if})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{arg}_1 : \beta_1\ \ldots\ \Gamma \vdash \mathit{arg}_n : \beta_n\\ \Gamma,\ \mathit{this} : B,\ \mathit{arg}_1 : \beta_1, \ldots, \mathit{arg}_n : \beta_n \vdash P : \beta\end{array}}{\Gamma, f : (\beta_1, \ldots, \beta_n) \xrightarrow{B} \beta \vdash \mathit{fun}\ f(\mathit{arg}_1, \ldots, \mathit{arg}_n) = \{P\}} \quad (\textsc{fun})$$

$$\frac{\Gamma \vdash E : (\beta_1, \ldots, \beta_n) \xrightarrow{B} \beta \qquad \Gamma \vdash E_1 : \beta_1\ \ldots\ \Gamma \vdash E_n : \beta_n}{\Gamma,\ \mathit{this} : B \vdash E(E_1, \ldots, E_n) : \beta} \quad (\textsc{funcall})$$

$$\frac{\Gamma \vdash E : (\beta_1, \ldots, \beta_n) \xrightarrow{B} \mathit{Unit} \qquad \Gamma \vdash E_1 : \beta_1\ \ldots\ \Gamma \vdash E_n : \beta_n}{\Gamma,\ \mathit{this} : B \vdash E(E_1, \ldots, E_n) : \mathit{Unit}} \quad (\textsc{proccall})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Ref}(\beta) \qquad \Gamma \vdash E_2 : \beta}{\Gamma \vdash E_1 = E_2} \quad (\textsc{assign})$$

$$\frac{}{\Gamma,\ \mathit{this} : l^{\uparrow}?(\beta) \vdash \texttt{receiveUp(l)} : \beta} \quad (\textsc{recvUp})$$

$$\frac{\Gamma \vdash E : \beta}{\Gamma,\ \mathit{this} : l^{\uparrow}!(\beta) \vdash \texttt{sendUp(l,E)}} \qquad (\textsc{sendUp})$$

$$\frac{\Gamma \vdash E : \beta}{\Gamma \vdash \mathit{return}\ E : \beta} \qquad (\textsc{return})$$

$$\frac{\Gamma \vdash E : \beta \qquad \beta\ \mathit{is\ either\ Int,\ Bool,\ Array,\ String}}{\Gamma \vdash \mathit{print}\ E} \qquad (\textsc{print})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Int} \qquad \Gamma \vdash E_2 : \beta}{\Gamma \vdash \mathit{array}(E_1, E_2) : \mathit{Array}(\beta)} \qquad (\textsc{array})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Array}(\beta) \qquad \Gamma \vdash E_2 : \mathit{Int} \qquad \Gamma \vdash E_3 : \beta}{\Gamma \vdash E_1[E_2] = E_3} \qquad (\textsc{arraySet})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Array}(\beta) \qquad \Gamma \vdash E_2 : \mathit{Int}}{\Gamma \vdash E_1[E_2] : \beta} \qquad (\textsc{arrayGet})$$

$$\frac{\Gamma \vdash E : \mathit{Array}(\beta)}{\Gamma \vdash \mathit{length}(E) : \mathit{Int}} \qquad (\textsc{arrayLength})$$

$$\frac{\Gamma \vdash E : \beta}{\Gamma \vdash \mathit{ref}(E) : \mathit{Ref}(\beta)} \qquad (\textsc{ref})$$

$$\frac{\Gamma \vdash E : \mathit{Ref}(\beta)}{\Gamma \vdash !E : \beta} \qquad (\textsc{bang})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Bool} \qquad \Gamma \vdash E_2 : \mathit{Bool}}{\Gamma \vdash E_1\ \mathit{and}\ E_2 : \mathit{Bool}} \qquad (\textsc{and})$$

$$\frac{\Gamma \vdash E_1 : \beta \qquad \Gamma \vdash E_2 : \beta \quad \mathit{with}\ \beta \in \{\mathit{Int}, \mathit{String}\}}{\Gamma \vdash E_1 + E_2 : \beta} \qquad (\textsc{add})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Int} \qquad \Gamma \vdash E_2 : \mathit{Int}}{\Gamma \vdash E_1 / E_2 : \mathit{Int}} \qquad (\textsc{div})$$

$$\frac{\Gamma \vdash E_1 : \mathit{Int} \qquad \Gamma \vdash E_2 : \mathit{Int}}{\Gamma \vdash E_1 - E_2 : \mathit{Int}} \qquad (\textsc{sub})$$

$$\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 \% E_2 : Int} \quad (\textsc{mod})$$

$$\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 * E_2 : Int} \quad (\textsc{mul})$$

$$\frac{\Gamma \vdash E : Bool}{\Gamma \vdash not\ E : Bool} \quad (\textsc{not})$$

$$\frac{\Gamma \vdash E_1 : \beta \qquad \Gamma \vdash E_2 : \beta}{\Gamma \vdash E_1 == E_2 : Bool} \quad (\textsc{equal})$$

$$\frac{\Gamma \vdash E_1 : \beta \qquad \Gamma \vdash E_2 : \beta}{\Gamma \vdash E_1 > E_2 : Bool} \quad (\textsc{greater})$$

$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash P}{\Gamma \vdash while(E)\ do\ \{P\}} \quad (\textsc{while})$$

$$\frac{\Gamma \vdash \beta}{\Gamma,\ id : \beta \vdash \diamond} \quad (\textsc{id})$$

$$\frac{}{\Gamma \vdash Bool} \quad (\textsc{t-bool})$$

$$\frac{}{\Gamma \vdash Int} \quad (\textsc{t-int})$$

$$\frac{}{\Gamma \vdash String} \quad (\textsc{t-string})$$

$$\frac{\Gamma \vdash \beta}{\Gamma \vdash Array(\beta)} \quad (\textsc{t-array})$$

$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \beta'}{\Gamma \vdash \beta \rightarrow \beta'} \quad (\textsc{t-arrow})$$

4.2 Typechecking Algorithm

The type inference algorithm takes as input a program P, a set of remote types declarations (in a typing environment Γ), an initially empty set of constraints on types R, and an initially empty set of apartness restrictions A. The algorithm outputs the type of program P, the typing environment Γ' where we can typecheck program P, a set of constraints R' and a set of apartness restrictions A'

$$\textbf{typecheck}(P,\Gamma,R,\ A) = (T,\Gamma',R',A')$$

We will now introduce some concepts necessary to understand how our algorithm works.

Definition 4.2.1 (Apartness)**.** *We say a label is apart from a behavioural type B, writing $l\#B$, if B has no occurrence of the label l.*

Definition 4.2.2 (Message Set)**.** *We define the message set on a behavioural type B, denoted as $\mathcal{M}(B)$, as follows:*

$$\begin{array}{lll}
\mathcal{M}(\emptyset) & ::= & \emptyset \\
\mathcal{M}(lp^d(\beta)) & ::= & lp^d \\
\mathcal{M}(B_1;B_2) & ::= & \mathcal{M}(B_1)\cup\mathcal{M}(B_2) \\
\mathcal{M}(B_1|B_2) & ::= & \mathcal{M}(B_1)\cup\mathcal{M}(B_2) \\
\mathcal{M}(\oplus\{l_1:B_1,\ldots,l_n:B_n\}) & ::= & \mathcal{M}(B_1)\cup\ldots\cup\mathcal{M}(B_n) \\
\mathcal{M}(\&\{l_1:B_1,\ldots,l_n:B_n\}) & ::= & \mathcal{M}(B_1)\cup\ldots\cup\mathcal{M}(B_n)
\end{array}$$

Definition 4.2.3 (Non Interference)**.** *We say two behavioural types B,B' do not interfere with each other, and denote as $B\#B'$, if the intersection of the set of message types occurring in B and B' is empty set,* i.e. $\mathcal{M}(B)\cap\mathcal{M}(B')=\emptyset$.

Definition 4.2.4 (Merge Relation)**.** *We define the merge of behavioural types as a ternary relation, $B = B_1 \bowtie B_2$, defined as:*

$$B = B \bowtie 0$$

$$(B \mid B');l\tau^d;B'' = B;l!^d;B_1 \bowtie B';l?^d;B_2 \qquad \textit{where } B''=B_1\bowtie B_2,\ l\#B'' \text{ and } B\#B'$$

$$(B \mid B');\oplus\{l_1:B_1'',\ldots,l_n:B_n''\};B'' = B;\&\{l_1:B_1,\ldots,l_n:B_n\};B_1''\bowtie B';\oplus\{l_1:B_1',\ldots,l_n:B_n'\};B_2''$$
$$\textit{where } B\#B' \text{ and } B''=B_1''\bowtie B_2'' \text{ and } B_i''=B_i\bowtie B_i',\ for\ i\in\{1,\ldots,n\}$$

$$(B \mid B') = B\bowtie B' \qquad \textit{where } B\#B'$$

Furthermore, the merge relation is commutative and associative.

Definition 4.2.5 (Constraint)**.** *A constraint is either an equation $E \doteq E'$ relating expressions E, E' (represented as $< E, E' >$ in a system of equations) or an apartness $l \# E$ (l can't occur in expression E). Expressions E are defined as follows:*

$$E \quad ::= \quad B \mid \beta \mid E \bowtie E' \mid K$$

where B is a behavioural type, β a basic type, and K a type variable as defined in Figure 4.1.

Notation 4.2.6 (Constraint Set)**.** *We denote a set of constraints as R.*

Definition 4.2.7 (Independent Constraint Sets)**.** *We say two constraints set, R and R', are independent of each other if their type variables set (that is, the set of all type variables occurring in a constraint set) are disjoint.*

Definition 4.2.8 (Solved and Unsolved Constraint)**.** *A solved constraint is an equation $K = E$ such that both $K \notin Var(E)$ and E is not a* merge constraint*. Therefore, an unsolved constraint is an equation that doesn't meet the previous conditions.*

Definition 4.2.9 (Substitution Application)**.** *We denote a substitution application as the application of a constraint set to a typing environment, $R(\Gamma)$, and define it as being the substitution of all occurrences of a type variable in the input environment with its corresponding type if, and only if, the constraint associated with the type variable is solved in R.*

Definition 4.2.10 (System of Equations)**.** *We denote a constraint set R as a system of equations where $\Sigma = \{\bowtie, |, ;, \oplus, \&\}$, $\Sigma_0 = \{M, \beta\}$ and $T_\Sigma(X)$ is the set of all terms formed from the function symbols and constants in Σ and Σ_0, respectively, as well as (type) variables from the set X.*

Definition 4.2.11 (Substitution)**.** *A substitution σ is a function from any variable in X to any term in $T_\Sigma(X)$. We write $[^T/_x]$ for the substitution that maps x to T and $D(\sigma)$ for the domain where a substitution is defined. Furthermore, we denote a substitution that respects a set of apartness restrictions, A, as σ^A.*

Definition 4.2.12 (Variables Set)**.** *We denote as $Var(t)$ the set of variables occurring in term t.*

Definition 4.2.13 (Unifiers)**.** *A substitution σ is called an unifier of an equation $< t, u >$ if $\sigma(t) = \sigma(u)$. A substitution σ is an unifier of a system R if it unifies each equation of R. The set of unifiers of a system R is denoted as $U(R)$.*

Definition 4.2.14 (Most General Unifier)**.** *A substitution σ is a most general unifier of a system R (written σ_R) if, and only if:*

1. $D(\sigma) \subseteq Var(R)$
2. $\sigma \in U(R)$
3. *For every $\theta \in U(R)$, $\theta = \sigma \circ \gamma$ for some substitution γ (or simply $\sigma <= \theta$)*

Definition 4.2.15 (Solved Equation). *An equation $< x,t >$ is solved in a system R if variable x does not occur anywhere else in system R, $x \notin Var(t)$, and t is not a term representing a merge constraint. Variable x is called a solved variable (likewise, an unsolved variable is a variable that occurs in R but it is not solved).*

Definition 4.2.16 (Solved System). *A system R is solved if all its equations are solved and all apartness restrictions are respected.*

Definition 4.2.17 (Transformations Rules). *Let R denote a system of equations, t a term, A an apartness constraint set, and T a type. We define the transformations rules, $R \Longrightarrow^A R'$ (if R does not violate any apartness constraint in A, then we can transform to a system R' that also complies with A), as follows:*

Trivial:

$$\{< t,t' >\} \cup R \Longrightarrow^A_{triv} R$$

where $t \equiv t'$

Variable Elimination:

$$\{< x,T >\} \cup R \Longrightarrow^A_{elem} \{< x,T >\} \cup R[^T/_x]$$

such that $x \notin Var(T)$

Merge Trivial:

$$\{< x, \bowtie (B) >\} \cup R \Longrightarrow^A_{merge\_trivial} \{< x,B >\} \cup R$$

Merge Inact:

$$\{< x, \bowtie (B_1,\ldots,B_{i-1},0,B_{i+1},\ldots,B_n) >\} \cup R \Longrightarrow^A_{merge\_inact}$$

$$\{< x, \bowtie (B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_n) >\} \cup R$$

Merge Parallel:

$$\{< x, \bowtie (B,\ldots,B_1|B_2,\ldots,B') >\} \cup R \Longrightarrow^A_{merge\_par}$$

$$\{< x, \bowtie (B,\ldots,B_1,B_2,\ldots,B) >\} \cup R$$

where $B_1 \neq 0$ and $B_2 \neq 0$.

Merge Sync:

$$\{< x, \bowtie(B_1,\ldots,B_i; l_1 p_1^{d_1}; B'_i,\ldots,\ B_j; l_2 p_2^{d_2}; B'_j,\ldots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_sync}$$
$$\{< x, (B_i \mid B_j); l_1 \tau^{d_1}; y >\} \cup \{< y, \bowtie(B_1,\ldots,B'_i,\ldots,B'_j,\ldots,B'_n) >\} \cup \sigma(R)$$

where $l_1 = l_2$, $d_1 = d_2$, p_1 is the opposite polarity of p_2, $A = A \cup \{l_1 \# y\}$, B_i and B_j don't interfere with each other nor any other B_k with $k \in \{1,\ldots,n\}$, and $\sigma = [^{(B_i \mid B_j); l_1 \tau^{d_1}; y}/_x]$.

Merge Choice Sync:

$$\{< x, \bowtie(B_1,\ldots,B_i; C; B'_i,\ldots,\ B_j; D; B'_j,\ldots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_sync2}$$
$$\{< x, (B_i \mid B_j); \oplus\{l_1 : y_1,\ldots,l_n : y_n\}; y >\} \cup$$
$$\{< y, \bowtie(B_1,\ldots,B'_i,\ldots,B'_j,\ldots,B'_n) >\} \cup$$
$$\{< y_1, \bowtie(B_{c1}, B'_{c1}) >\} \cup \ldots \cup \{< y_n, \bowtie(B_{cn}, B'_{cn}) >\} \cup \sigma(R)$$

where $C = \&\{l_1 : B_{c1},\ \ldots,\ l_n : B_{cn}\}$ and $D = \oplus\{l_1 : B'_{c1},\ \ldots,\ l_n : B'_{cn}\}$, and B_i and B_j don't interfere with each other nor any other B_k with $k \in \{1,\ldots,n\}$, and $\sigma = [^{(B_i \mid B_j); \oplus\{l_1 : y_1,\ldots,l_n : y_n\}; y}/_x]$.

Remark 4.2.18. *It follows from the transformation rules definition that if we obtain a solution σ to a system R by applying these rules then it must be that the substitution respects the apartness set generated by the application of the rules, so $\sigma = \sigma^A$.*

Definition 4.2.19 (System's Solution)**.** *We say σ is a solution to a system R if, and only if, it is possible to apply transformations rules such that the resulting system R' is a solved system and σ is its most general unifier. If it is not possible to apply a transformation rule and the system is still not solved then no solution is possible for the system and the procedure fails.*

Lemma 4.2.20 (Substitution Preservation)**.** *If we can obtain a system R' from using a transformation rule on a system R then they both share the same set of unifiers:*

$$R \Longrightarrow^{A} R' \quad then \quad U(R) = U(R')$$

Proof. Immediate for the trivial rule.The rules merge trivial, merge inact and merge parallel only rewrite an equation E to an equivalent equation E' and, therefore, $\sigma(E) = \sigma(E')$ for any unifier σ of the systems R, R'.

For the elimination rule we have that $\{< x, T >\} \cup R \Longrightarrow^{A}_{elem} \{< x, T >\} \cup \sigma(R)$, where $\sigma = [^T/_x]$. So for any substitution θ, if $\theta(x) = \theta(T)$, we have that $\theta = \sigma \circ \theta$ where θ differs from $\sigma \circ \theta$ only at x, but $\theta(x) = \theta(T) = \sigma \circ \theta(x)$. Therefore

$\theta \in U(R \cup \{< x, T >\})$
$\Leftrightarrow \theta(x) = \theta(T)$ *and* $\theta \in U(R)$
$\Leftrightarrow \theta(x) = \theta(T)$ *and* $\sigma \circ \theta \in U(R)$
$\Leftrightarrow \theta(x) = \theta(T)$ *and* $\theta \in U(\sigma(R))$
$\Leftrightarrow \theta \in U(\sigma(R) \cup \{< x, T >\})$

For the merge sync rule we have that $\{< x, \bowtie (\ldots) >\} \cup R \Longrightarrow^{A}_{merge} \{< x, (B|B'); l\tau^{d}; y >\} \cup \{< y, \bowtie (\ldots) >\} \cup \sigma(R) \cup \{l\#y\}$, where $\sigma = [^{(B|B'); l\tau^{d}; y}/_{x}]$ and since adding constraints to R' will make the unifier of R' contain solutions to those constraints and still unify system R then we use the same reasoning we used for the variable elimination rule. Likewise for the choice transformation rule. □

Theorem 4.2.21 (Unification Soundness). *If $R \Longrightarrow^{A*} R'$ and R' is a solved system, then the most general unifier of the solved system, $\sigma_{R'}$, is also a unifier of the system R,* i.e. $\sigma_{R'} \in U(R)$.

Proof. By lemma 4.2.20, we have that each application of a transformation rule preserves the set of unifiers of the system ergo when we obtain the solved system R' through successive applications of the rules we have that the set of unifiers of R' unifies R and, by definition of most general unifier 4.2.14, $\sigma_{R'} \in U(R') = U(R)$.

□

Definition 4.2.22 (Term Size). *We define the size of a term as the number of labels occurring in the term.*

Theorem 4.2.23 (Unification Completeness). *For a substitution $\theta \in U(R)$, any application of transformation rules to R*

$$R \Longrightarrow^{A} R_1 \Longrightarrow^{A} R_2 \Longrightarrow^{A} R_3 \Longrightarrow^{A} \ldots$$

terminates in a system R' that is solved and its unifier σ is such that $\sigma_{R'} <= \theta$.

Proof. Let $< m, n >$ be a pair such that n is the number of variables unsolved in R and m the sum of the sizes of each term in R, then the lexicographic order of such pairs is a well-founded relation. We then prove that every transformation sequence terminates since each transformation results in a system where the pair $< m, n >$ is smaller under the lexicographic ordering. The Trivial rule does not change any of the values, the Merge rules either decreases or do not alter the value of m, and the rule Variable Elimination decreases n.

We have then proved that any sequence of transformation rules terminate and the system obtained, R', is one to which no further rules can be applied. Then, we have that for some substitution $\theta \in U(R)$, by Lemma 4.2.20, it is also true that $\theta \in U(R')$, in particular since no further transformation can be done then it must be that R' is in solved form. Furthermore, since $\theta \in U(R')$ we have $\sigma_{R'} <= \theta$.

□

The unification algorithm receives a constraint set R, whose constraints represent a system of equations, and an initially empty apartness set A as input. The algorithm consists in the manipulation of the system of equations through the application of transformation rules until it is in solved form, this is a standard technique [24, 12]. During constraint solving, matching labels may need to be synchronised (for *e.g.*, when we have a merge constraint on two dual labels). To ensure linearised usage, we have introduced a new kind of constraint (checked on each transformation step) to state that a label cannot occur in a given type. We denote this as an apartness restriction $l\#B$ (added to the apartness set A). Obviously, if at any moment a step can not be executed the algorithm aborts since the program must be ill-typed.

The constraints generated have the form $< E, E' >$ according to the syntax presented in Definition 4.2.5. We have standard constraints like $< x, T >$ and $< x, y >$, where the former states that a type variable x has type T (either behavioural type B or a basic type β), whilst the latter imposes type equality on the solved types for the type variables x and y. In the unification algorithm, these are treated using the standard transformation rules for variable elimination and type equality, a solvable system terminates in a system in solved form, that corresponds to a substitution.

We also introduce a merge constraint on types of the form $< x, \bowtie (B, B') >$, that constrains type variable x to be a composition of behavioural types B and B' (this operation is defined by a *merge relation* in Definition 4.2.4, adapted from [4]). Merge constraints are necessary to represent the type of a parallel composition of code (where synchronisation can happen) or when we invoke a service via the join primitive (since we merge the behaviours of the invoked service with the client's) and thus we need to be able to represent the *merge* of all behaviour in the composition such that casual ordering is kept and interleaves are avoided unless there is a synchronisation: this way, the most general (less serialised) behaviour is computed. Merge constraints are solved using a set of transformation rules that represent the *merge relation* on behavioural types.

Definition 4.2.24 (Constraint Solving Algorithm)**.** *Let R be a constraint set, whose constraints represent a system of equations, and A an initially empty apartness set.*

```
**resolve**(R, A) $\triangleq$
  **let** R = {$E \doteq E'$} ∪ R' **then**
    verifyConsistency(A)
    **case** {$T \doteq T'$} **then**
      **if** equals(T, T') **then**
```

```
        **resolve**$(R', A)$
    **case** $\{K \doteq T\}$ **then**
        $R'' = R'[^K/_T] \cup \{K = T\}$ and **resolve**$(R'', A)$
    **case** $\{K \doteq \bowtie(B)\}$ **then**$\}$
        $R'' = \{K \doteq B\} \cup R'$ and **resolve**$(R'', A)$
    **case** $\{K \doteq \bowtie(B, \ldots, B_1 | B_2, \ldots, B_n)\}$ **then**$\}$
        $R'' = \{K \doteq \bowtie(B_1, \ldots, B_1, B_2, \ldots, B_m)\} \cup R'$ and **resolve**$(R'', A)$
    **case** $\{K \doteq \bowtie(B_1, \ldots, B_{i-1}, 0, B_{i+1}, \ldots, B_m)\}$ **then**
        $R'' = \{K \doteq \bowtie(B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_m)\} \cup R'$ and **resolve**$(R'', A)$
    **case** $\{K \doteq \bowtie(B_1, \ldots, B_i;l_1 p_1^{d_1};B'_i, \ldots, B_j;l_2 p_2^{d_2};B'_j, \ldots, B_n)\}$ **then**
      **if** $(B_i \# B_j$ and $l_1 = l_2$ and $d_1 = d_2$ and $p_1$ is the opposite polarity of $p_2)$ **then**
        **let** $\sigma = [^{(B_i|B_j);l_1\tau^{d_1};K'}/_K]$
        $R'' = \{K = (B_i|B_j);l_1\tau^{d_1};K'\} \cup \{K' \doteq \bowtie(B_1, \ldots, B'_i, \ldots, B'_j, \ldots, B'_n)\} \cup \sigma(R')$
        and **resolve**$(R'', A \cup \{l_1 \# K'\})$
    **case** $\{K \doteq \bowtie(B_1, \ldots, B_i;C;B'_i, \ldots, B_j;D;B'_j, \ldots, B_n)\}$ **then**
      **if**$(C = \&\{l_1 \colon B_{c1}, \ldots, l_n \colon B_{cn}\}$ and $D = \oplus\{l_1 \colon B'_{c1}, \ldots, l_n \colon B'_{cn}\}$ and $B_i \# B_j)$ **then**
        **let** $\sigma = [^{(B_i|B_j);\oplus\{l_1:K_1,\ldots,l_n:K_n\};K'}/_K]$
        $R'' = \{K = (B_i|B_j);\oplus\{l_1 \colon K_1, \ldots, l_n \colon K_n\};K'\} \cup \{K_i \doteq \bowtie(B_{ci}, B'_{ci})\}$
                $\cup \{K' \doteq \bowtie(B_1, \ldots, B'_i, \ldots, B'_j, \ldots, B_n)\}\} \cup \sigma(R')$
        and **resolve**$(R'', A)$
```

Remark 4.2.25 (Verify consistency operation)**.** *The operation* `verifyConsistency(A)` *checks if any apartness constraint is violated.*

Theorem 4.2.26 (Constraint Solving Algorithm Soundness)**.** *The constraint solving algorithm is sound w.r.t. the unification problem for a system composed by the equations in R with the set of apartness constraints A.*

Proof. From the definition of the constraint solving algorithm 4.2.24 where we have that the substitution obtained is a most general unifier for a system formed by the restrictions in R after applying transformation rules and, henceforth, it is equivalent to the unification problem where we have proved its soundness. □

Lemma 4.2.27 (Valid Substitution)**.** *The substitution θ obtained by the resolve algorithm is said to be valid if the following is true:*

$$\texttt{resolve}(\mathtt{R},\mathtt{A}) = (\theta, \mathtt{A}') \quad \Rightarrow \quad \forall_{\mathtt{E} \doteq \mathtt{E}'} \in \mathtt{R} \quad \theta(\mathtt{E}) = \theta(\mathtt{E}')$$

Proof. Immediate since we know that θ is a unifier for the system represent by R and thus, be definition of a system of equations' unifier 4.2.13, $\forall_{E \doteq E'} \in R \quad \theta(E) = \theta(E')$ holds. □

The typechecking algorithm consists of the following steps. First, it transverses the abstract syntax tree, applying typing rules backward if possible. Whenever a type needs to be inferred, a constraint is generated and added to the set R. Finally, the constraints are solved with the unification algorithm (*i.e.* the constraint solving algorithm) previously presented.

Definition 4.2.28 (Typechecking Algorithm)**.** *Let* Γ *be a typing environment initially containing a set of remote types declarations, R an initially empty set of constraints on types, and A an initially empty set of apartness restrictions.*

```
**typecheck**(**invoke** $s$ **in** $n$ **as** $\{ P \}, \Gamma, R, A) \triangleq$
    **let** $(\Gamma' \cup \{this: B\}, R', A') =$ **typecheck**$(P, \Gamma, R, A)$ and
    **let** $(R'', A'') =$ **resolve**$(R', A')$ and
    **let** $\{n{:}[s](B_1)\} \in \Gamma$, verify $B_1 = \overline{\downarrow B}$ **then**
        **return** $(\Gamma' \cup \{ this: loc(\uparrow B)\}, R'', A'')$

**typecheck**$(P_1; P_2, \Gamma, R, A) \triangleq$
    **let** $(\Gamma_1 \cup \{this: B_1\}, R', A') =$ **typecheck**$(P_1, \Gamma, R, A)$ and
    **let** $(\Gamma_2 \cup \{this: B_2\}, R'', A'') =$ **typecheck**$(P_2, \Gamma, R', A')$ and
        **return** $(\Gamma_1 \cup \Gamma_2 \cup \{this: B_1;B_2\}, R'', A'')$

**typecheck**$(P_1 \parallel \ldots \parallel P_n, \Gamma, R, A) \triangleq$
    **let** $B = 0$ and
    **let** $\Gamma_r$ is empty and
    for each $P_i$ where $i \in \{1, \ldots, n\}$ **do**
        **let** $(\Gamma_i \cup \{this: B_i\}, R_i, A_i) =$ **typecheck**$(P_i, \Gamma, R, A)$ and
            $B = B \bowtie B_i$ and $\Gamma_r = \Gamma_r \cup \Gamma_i$
    for some fresh $K$, **let** $R' = (\bigcup_{i \in \{1 \ldots n\}} R_i) \cup \{K \doteq B\}$
    **return** $(\Gamma_r \cup \{this: K\}, R', \bigcup_i A_i)$

**typecheck**(**send**$(l, E), \Gamma, R, A) \triangleq$
    **let** $(\beta, \Gamma', R', A') =$ **typecheck**$(E, \Gamma, R, A)$ and
        **return** $(\Gamma \cup \{this: l!(\beta)\}, R', A')$

**typecheck**(**receive**$(l), \Gamma, R, A) \triangleq$
    for some fresh $K$
        **return** $(K, \Gamma \cup \{this: l?(K)\}, R, A)$

**typecheck**(**sendUp**$(l, E), \Gamma, R, A) \triangleq$
    **let** $(\beta, \Gamma', R', A') =$ **typecheck**$(E, \Gamma, R, A)$ and
        **return** $(\Gamma \cup \{this: l^{\uparrow}!(\beta)\}, R', A')$

**typecheck**(**receiveUp**$(l), \Gamma, R, A) \triangleq$
    for some fresh $K$
        **return** $(K, \Gamma \cup \{this{:}l^{\uparrow}?(K)\}, R, A)$
```

```
typecheck(site $n$ { $P$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\Gamma' \cup$ {this: $B$}, $R'$, $A'$) = typecheck($P$, $\Gamma$, $R$, $A$) and
    let ($R''$, $A''$) = resolve($R'$, $A'$) then
    return ($\Gamma' \cup$ {$n$: $B$}, $R''$, $A''$)

typecheck(def $s$ as { $P$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\Gamma' \cup$ {this: $B$}, $R'$, $A'$) = typecheck($P$, $\Gamma$, $R$, $A$) and
    return ($\Gamma' \cup$ {this: $[s](\downarrow B)$;loc($\uparrow B$)}, $R'$, $A'$)

typecheck(join $s$ in $n$ as { $P$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\Gamma' \cup$ {this: $B$}, $R'$, $A'$) = typecheck($P$, $\Gamma$, $R$, $A$) and
    let {$n$: $[s](B_1)$} $\in \Gamma$ then
    for some fresh $K$, $R'' = R' \cup \{K \doteq B \bowtie B_1\}$ do
    return($\Gamma' \cup$ {this: $K$}, $R''$, $A'$)

typecheck(let $x = E$ in { $P$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\beta$, $\Gamma \cup$ {this: $B_1$}, $R'$, $A'$) = typecheck($E$, $\Gamma$, $R$, $A$) and
    let ($\Gamma_2 \cup \{x{:}\beta\} \cup$ {this: $B_2$}, $R''$, $A''$) = typecheck($P$, $\Gamma \cup \{x{:}\beta\}$, $R'$, $A'$) and
    return ($\Gamma_2 \cup$ {this: $B_1$;$B_2$}, $R''$, $A''$)

typecheck(if($E$) then { $E_1$ } else { $E_2$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\beta_E$, $\Gamma'$, $R'$, $A'$) = typecheck($E$, $\Gamma$, $R$, $A$) and
    if $\beta_E$ is type variable $K$ then
        $R' = R' \cup \{K \doteq Bool\}$
    else verify $\beta_E = Bool$
    let ($\beta$, $\Gamma \cup$ {this: $B_{E1}$}, $R''$, $A''$) = typecheck($E_1$, $\Gamma$, $R'$, $A'$) and
    let ($\beta$, $\Gamma \cup$ {this: $B_{E2}$}, $R'''$, $A''$) = typecheck($E_2$, $\Gamma$, $R''$, $A''$) then
    if $B_{E1}$ and $B_{E2} \neq$ InactionType then
        let $B_{E1} = l_1;B_1$ and $B_{E2} = l_2;B_2$
        return ($\beta$, $\Gamma \cup$ {this: $\oplus\{l_1{:}\ l_1;B_1,\ l_2{:}\ l_2;B_2\}$ }, $R'''$, $A'''$)
    else return ($\beta$, $\Gamma$, $R'''$, $A'''$)

typecheck(if($E$) then { $P_1$ } else { $P_2$ }, $\Gamma$, $R$, $A$) $\triangleq$
    let ($\beta_E$, $\Gamma'$, $R'$, $A'$) = typecheck($E$, $\Gamma$, $R$, $A$) and
    if $\beta_E$ is type variable $K$ then
        $R' = R' \cup \{K \doteq Bool\}$
    else verify $\beta_E = Bool$
    let ($\Gamma' \cup$ {this: $B_{E1}$}, $R''$, $A''$) = typecheck($P_1$, $\Gamma$, $R'$, $A'$) and
    let ($\Gamma'' \cup$ {this: $B_{E2}$}, $R''$', $A''$) = typecheck($P_2$, $\Gamma$, $R''$, $A''$) then
    if $B_{E1}$ and $B_{E2} \neq$ InactionType then
        let $B_{E1} = l_1;B_1$ and $B_{E2} = l_2;B_2$
        return ($\Gamma' \cup \Gamma'' \cup$ {this: $\oplus\{l_1{:}\ l_1;B_1,\ l_2{:}\ l_2;B_2\}$ }, $R'''$, $A'''$)
    else return ($\Gamma' \cup \Gamma''$, $R'''$, $A'''$)
```

```
$\textbf{typecheck}(\textbf{select}\ \{l_1{:}\ P_1, \ldots, l_n{:}\ P_n\}, \Gamma, R, A) \triangleq$
    $\textbf{let}\ (\Gamma_i \cup \{this{:}\ B_i\}, R_i, A_i) = \textbf{typecheck}(P_i, \Gamma, R, A),\ for\ i \in \{1, \ldots, n\}\ and$
    $\textbf{return}\ (\bigcup_i \Gamma_i \cup \{this{:}\ \&\{l_1{:}\ B_1, \ldots, l_n{:}\ B_n\}\ \}, \bigcup_i R_i, \bigcup_i A_i)$

$\textbf{typecheck}(\textbf{switch}\ \{\ \textbf{case}\ (E_1)\ \textbf{do}\ l_1{:}\ P_1, \ldots, \textbf{case}\ (E_n)\ \textbf{do}\ l_n{:}\ P_n, \textbf{default do}\ l_i{:}\ P_d\ \}, \Gamma, R, A) \triangleq$
    $\textbf{let}\ (\beta_{Ei}, \Gamma', R'_i, A'_i) = \textbf{typecheck}(E_i, \Gamma, R, A)\ and$
    $\textbf{if}\ \beta_{Ei}\ is\ type\ variable\ K\ \textbf{then}$
        $R'_i = R'_i \cup \{K \doteq Bool\}$
    $\textbf{else}\ verify\ \beta_{Ei} = Bool$
    $\textbf{let}\ (\Gamma_i \cup \{this{:}\ B_i\}, R_i, A_i) = \textbf{typecheck}(P_i, \Gamma, R'_i, A'_i),\ for\ i \in \{1, \ldots, n\}\ and$
    $\textbf{if}\ l_i \in labels\_map\ \textbf{then}$
        $verify\ B_i = labels\_map.get(l_i)$
    $\textbf{else}\ labels\_map.put(l_i, B_i)$
    $\textbf{let}\ (\Gamma_d \cup \{this{:}\ B_d\}, R_d, A_d) = \textbf{typecheck}(P_d, \Gamma, R, A)\ \textbf{then}$
    $\textbf{let}\ m = labels\_map.size()\ and$
    $\textbf{let}\ B_1, \ldots, B_m\ the\ respective\ labels\_map.get(l_j),\ j \in \{1, \ldots, m\}\ \textbf{then}$
    $\textbf{return}\ (\bigcup_i \Gamma_i \cup \Gamma_d \cup \{this{:} \oplus\{\ l_1{:}\ B_1, \ldots, l_m{:}\ B_m\}\ \}, \bigcup_i R_i \cup R_d, \bigcup_i A_i \cup A_d)$

$\textbf{typecheck}(\textbf{fun}\ f(arg_1, \ldots, arg_n) = \{\ P\ \}, \Gamma, R, A) \triangleq$
    $\textbf{let}\ (K_i, \Gamma, R, A) = \textbf{typecheck}(arg_i, \Gamma, R, A),\ for\ i \in \{1, \ldots, n\},\ K_i\ fresh\ and$
    $\textbf{let}\ (\beta, \Gamma_2 \cup \{arg_1{:}\ K_1, \ldots, arg_n{:}\ K_n\} \cup \{this{:}\ B\}, R', A') =$
          $\textbf{typecheck}(P, \Gamma \cup \{arg_1{:}\ K_1, \ldots, arg_n{:}\ K_n\}, R, A),\ \textbf{then}$
    $\textbf{return}\ (\Gamma_2 \cup \{f{:}\ (K_1, \ldots, K_n \rightarrow^B \beta)\}, R', A')$

$\textbf{typecheck}(E(E_1, \ldots, E_n), \Gamma, R, A) \triangleq$
    $\textbf{let}\ (\ (\beta_1, \ldots, \beta_n) \rightarrow^B \beta, \Gamma, R, A) = \textbf{typecheck}(E, \Gamma, R, A),\ \textbf{then}$
    $\textbf{let}\ (\beta_{Ei}, \Gamma, R_i, A_i) = \textbf{typecheck}(E_i, \Gamma, R, A),\ for\ i \in \{1, \ldots, n\},\ and$
    $verify\ \beta_i = \beta_{Ei}$
    $\textbf{return}\ (\beta, \Gamma \cup \{this{:}\ B\}, \bigcup_i R_i, \bigcup_i A_i)$

$\textbf{typecheck}(E(E_1, \ldots, E_n), \Gamma, R, A) \triangleq$
    $\textbf{let}\ (\ (\beta_1, \ldots, \beta_n) \rightarrow^B Unit, \Gamma, R, A) = \textbf{typecheck}(E, \Gamma, R, A),\ \textbf{then}$
    $\textbf{let}\ (\beta_{Ei}, \Gamma, R_i, A_i) = \textbf{typecheck}(E_i, \Gamma, R, A),\ for\ i \in \{1, \ldots, n\},\ and$
    $verify\ \beta_i = \beta_{Ei}$
    $\textbf{return}\ (\Gamma \cup \{this{:}\ B\}, \bigcup_i R_i, \bigcup_i A_i)$

$\textbf{typecheck}(E_1 = E_2, \Gamma, R, A) \triangleq$
    $\textbf{let}\ (Ref(\beta_1), \Gamma, R, A) = \textbf{typecheck}(E_1, \Gamma, R, A),\ and$
    $\textbf{let}\ (\beta_2, \Gamma, R', A') = \textbf{typecheck}(E_2, \Gamma, R, A)\ \textbf{then}$
    $\textbf{if}\ \beta_1\ is\ a\ type\ variable\ K_1\ \textbf{then}$
        $R' = R' \cup \{K_1 \doteq \beta_2\}$
    $\textbf{if}\ \beta_2\ is\ a\ type\ variable\ K_2\ \textbf{then}$
```

```
        $R' = R' \cup \{K_2 \doteq \beta_1\}$
    else verify $\beta_1 = \beta_2$
    return $(\Gamma, R', A')$

typecheck(return $E, \Gamma, R, A) \triangleq$
    let $(\beta, \Gamma, R', A')$ = typecheck($E, \Gamma, R, A$) then
    return $(\beta, \Gamma, R', A')$

typecheck(print $E, \Gamma, R, A) \triangleq$
    let $(\beta, \Gamma, R', A')$ = typecheck($E, \Gamma, R, A$) then
    return $(\Gamma, R', A')$

typecheck(array($E_1, E_2$), $\Gamma, R, A) \triangleq$
    let $(\beta_1, \Gamma, R', A')$ = typecheck($E_1, \Gamma, R, A$) and
    if $\beta_1$ is type variable $K$ then
        $R' = R' \cup \{K \doteq Int\}$
    else verify $\beta_1 = Int$
    let $(\beta, \Gamma, R'', A'')$ = typecheck($E_2, \Gamma, R', A'$) then
    return (Array($\beta$), $\Gamma, R'', A''$)

typecheck($E_1[E_2] = E_3, \Gamma, R, A) \triangleq$
    let (Array($\beta$), $\Gamma, R, A$) = typecheck($E_1, \Gamma, R, A$) and
    let $(\beta_1, \Gamma, R', A')$ = typecheck($E_2, \Gamma, R, A$) and
    if $\beta_1$ is type variable $K_1$ then
        $R' = R' \cup \{K_1 \doteq Int\}$
    else verify $\beta_1 = Int$
    let $(\beta_2, \Gamma, R'', A'')$ = typecheck($E_3, \Gamma, R', A'$) then
    if $\beta_2$ is type variable $K_2$ then
        $R'' = R'' \cup \{K_2 \doteq \beta\}$
    else verify $\beta_2 = \beta$
    return $(\Gamma, R'', A'')$

typecheck($E_1[E_2], \Gamma, R, A) \triangleq$
    let $(\beta_0, \Gamma, R', A')$ = typecheck($E_1, \Gamma, R, A$) and
    if $\beta_0$ is type variable $K_0$ then
        $R' = R' \cup \{K_0 \doteq Array(K')\}$ and $\beta = K'$ for some fresh type variable $K'$
    else verify $\beta_0$ = Array($\beta$) and
    let $(\beta_1, \Gamma, R'', A'')$ = typecheck($E_2, \Gamma, R', A'$) then
    if $\beta_1$ is type variable $K$ then
        $R'' = R'' \cup \{K \doteq Int\}$
    else verify $\beta_1 = Int$
    return $(\beta, \Gamma, R'', A'')$
```

```
**typecheck**(lenght(E), $\Gamma$, R, A) $\triangleq$
    **let** $(\beta_0, \Gamma, R', A')$ = **typecheck**$(E, \Gamma, R, A)$ **then**
    **if** $\beta_0$ is type variable $K_0$ **then**
        $R' = R' \cup \{K_0 \doteq Array(K')\}$ and $\beta = K'$ for some fresh type variable $K'$
    **else** verify $\beta_0 = Array(\beta)$ and
    **return** $(Int, \Gamma, R', A')$

**typecheck**(ref(E), $\Gamma$, R, A) $\triangleq$
    **let** $(\beta, \Gamma, R', A')$ = **typecheck**$(E, \Gamma, R, A)$ **then**
    **return** $(Ref(\beta), \Gamma, R', A')$

**typecheck**(!E, $\Gamma$, R, A) $\triangleq$
    **let** $(\beta', \Gamma, R', A')$ = **typecheck**$(E, \Gamma, R, A)$ **then**
    **if** $\beta'$ is a type variable $K$ **then**
        $R' = R' \cup \{K \doteq Ref(K')\}$ and $\beta = K'$ for a fresh type variable $K'$
    **else** verify $\beta' = Ref(\beta)$
    **return** $(\beta, \Gamma, R', A')$

**typecheck**($E_1$ and $E_2$, $\Gamma$, R, A) $\triangleq$
    **let** $(\beta_1, \Gamma, R', A')$ = **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'')$ = **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Bool\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Bool\}$
    **else** verify $\beta_1 = \beta_2 = Bool$
    **return** $(Bool, \Gamma, R'', A'')$

**typecheck**($E_1 + E_2$, $\Gamma$, R, A) $\triangleq$
    **let** $(\beta_1, \Gamma, R', A')$ = **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'')$ = **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Int\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Int\}$
    **else** verify $\beta_1 = \beta_2 = Int$
    **return** $(Int, \Gamma, R'', A'')$

**typecheck**($E_1 / E_2$, $\Gamma$, R, A) $\triangleq$
    **let** $(\beta_1, \Gamma, R', A')$ = **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'')$ = **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
```

```
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Int\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Int\}$
    **else** verify $\beta_1 = \beta_2 = Int$
    **return** $(Int, \Gamma, R'', A'')$

**typecheck**$(E_1 - E_2, \Gamma, R, A) \triangleq$
    **let** $(\beta_1, \Gamma, R', A') =$ **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'') =$ **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Int\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Int\}$
    **else** verify $\beta_1 = \beta_2 = Int$
    **return** $(Int, \Gamma, R'', A'')$

**typecheck**$(E_1 \% E_2, \Gamma, R, A) \triangleq$
    **let** $(\beta_1, \Gamma, R', A') =$ **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'') =$ **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Int\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Int\}$
    **else** verify $\beta_1 = \beta_2 = Int$
    **return** $(Int, \Gamma, R'', A'')$

**typecheck**$(E_1 * E_2, \Gamma, R, A) \triangleq$
    **let** $(\beta_1, \Gamma, R', A') =$ **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'') =$ **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\} \cup \{K_1 \doteq Int\}$
    **if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\} \cup \{K_2 \doteq Int\}$
    **else** verify $\beta_1 = \beta_2 = Int$
    **return** $(Int, \Gamma, R'', A'')$

**typecheck**$(not\ E, \Gamma, R, A) \triangleq$
    **let** $(\beta, \Gamma, R', A') =$ **typecheck**$(E, \Gamma, R, A)$ **then**
    **if** $\beta$ is a type variable $K$ **then**
        $R' = R' \cup \{K \doteq Bool\}$
    **else** verify $\beta = Bool$
    **return** $(Bool, \Gamma, R', A')$
```

```
**typecheck**$(E_1 == E_2, \Gamma, R, A) \triangleq$
    **let** $(\beta_1, \Gamma, R', A')$ = **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'')$ = **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\}$
    **else if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\}$
    **else** verify $\beta_1 = \beta_2$
    **return** $(Bool, \Gamma, R'', A'')$
```

```
**typecheck**$(E_1 > E_2, \Gamma, R, A) \triangleq$
    **let** $(\beta_1, \Gamma, R', A')$ = **typecheck**$(E_1, \Gamma, R, A)$ and
    **let** $(\beta_2, \Gamma, R'', A'')$ = **typecheck**$(E_2, \Gamma, R', A')$ **then**
    **if** $\beta_1$ is a type variable $K_1$ **then**
        $R'' = R'' \cup \{K_1 \doteq \beta_2\}$
    **else if** $\beta_2$ is a type variable $K_2$ **then**
        $R'' = R'' \cup \{K_2 \doteq \beta_1\}$
    **else** verify $\beta_1 = \beta_2$
    **return** $(Bool, \Gamma, R'', A'')$
```

```
**typecheck**(**while** $(E)$ **do** $\{ P \}, \Gamma, R, A) \triangleq$
    **let** $(\beta, \Gamma, R', A')$ = **typecheck**$(E, \Gamma, R, A)$ **then**
    **if** $\beta$ is a type variable $K$ **then**
        $R' = R' \cup \{K \doteq Bool\}$
    **else** verify $\beta = Bool$
    **let** $(\Gamma, R'', A'')$ = **typecheck**$(P, \Gamma, R', A')$ **then**
    **return** $(\Gamma, R'', A'')$
```

Lemma 4.2.29 (Decidability)**.** *The typechecking algorithm is decidable.*

Proof. The algorithm is syntax driven so the only point where it could diverge would be in the application of the unification algorithm to solve constraints. But we know by Theorem 4.2.23 that unification always terminates. Therefore the typechecking algorithm always terminates. □

Lemma 4.2.30 (Determinism)**.** *The typechecking algorithm is deterministic.*

Proof. The algorithm is syntax driven so the only point where it could be non-deterministic would be in the application of the unification algorithm to solve constraints. But the application of transformation rules is deterministic therefore the typechecking algorithm is deterministic. □

Lemma 4.2.31 (Principal Typing)**.** *The typechecking algorithm outputs a principal typing.*

Proof. This follows from the unification algorithm always returning the most general unifier to the system of equations. □

Lemma 4.2.32 (Monotonicity)**.** *The typechecking algorithm is a monotonic function:*

If **typecheck**$(P, \Gamma, R,\ A) = (T, \Gamma', R',\ A')$ *then* **typecheck**$(P, \Gamma', R',\ A') = (T, \Gamma', R',\ A')$

Lemma 4.2.33. *The relation between the constraint set used to typecheck a program and the resulting constraint set is the following:*

$$\textbf{typecheck}(P, \Gamma, R,\ A) = (T, \Gamma', R',\ A') \quad \Rightarrow \quad R' = \sigma^A(R) \cup N$$

where σ^A is a unifier of R respecting A and N a constraint set representing the new constraints obtained by typechecking program P.

Proof. By induction on the execution of the typecheck algorithm with program P. Complete proof at A.1, page 85.

□

Theorem 4.2.34 (Soundness of Typechecking Algorithm)**.** *Let P be a program, Γ, Γ' type environments , T a type, A a set of apartness constraints, and R a constraint set.*

If **typecheck**$(P, \Gamma, \emptyset, \emptyset) = (T, \Gamma', R, A)$*. Then $R(\Gamma') \vdash P : R(T)$ and there is a substitution θ such that $\theta(R(\Gamma')) \vdash P : \theta(R(T))$*

Proof. By induction on the execution of the typecheck algorithm with program P. Complete proof at A.2, page 89.

□

Theorem 4.2.35 (Completeness of Typechecking Algorithm)**.** *Let P be a program, Γ a typing context, Γ' a typing context containing only type declarations of remote services, and T a type*

If $\Gamma \vdash P : T$. Then **typecheck**$(P, \Gamma', \emptyset, \emptyset) = (T', \Gamma'', R', A')$ *and there is a substitution θ such that $\theta(\mathtt{R}'(\mathtt{T}')) = \mathtt{T}$ and $\theta(\mathtt{R}'(\Gamma'')) = \Gamma$ and $\theta(\mathtt{R}'(\Gamma')) \subseteq \Gamma$*

Proof. By induction on the structure of P. Complete proof at A.3, page 100

□

4.3 Examples

We now show how our typechecking algorithm works through its application to some examples.

4.3.1 The Weather Forecast Service

As input we have

> **P = code of weather forecast's site as shown in Figure** 1.2**, with remote types**
> **Γ = { #WeatherStation:[#weatherReport](#getReport?(_);#report!(String)) }**
> **R = A = $\emptyset$**

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking the join primitive, a type variable **x** is created as well as a constraint to represent the merge of the primitive's body's behaviour with the behaviour of the invoked service, **<x, $\bowtie$(#getReport!(String), #getReport?(_);#report!(String))>** that is added to R;
3. The constraint is resolved upon the typecheck of the site primitive and the algorithm terminates.

The algorithm outputs **(Γ $\cup$ #WeatherSite:**[**#weatherForecast**](**#location?(z);B), R, A$'$)** where **B** is the type obtained by the unification algorithm when solving the constraint on **x**, and **A$'$** the resulting apartness set.

In step 3 the unification algorithm is called with the following input:

> **R = { <x, $\bowtie$(#getReport!(String), #getReport?(_);#report!(String))> }**
> **A = $\emptyset$**

So **resolve**(R, A) takes the following steps:

> **<x, $\bowtie$(#getReport!(String), #getReport?(_);#report!(String))>** $\Longrightarrow^{\mathbf{A}}_{\mathbf{merge\_sync}}$
>
> **<x, #getReportτ(String);y> $\cup$ <y, $\bowtie$($\emptyset$, #report!(String))>** $\Longrightarrow^{\mathbf{A'}}_{\mathbf{merge\_inact}}$
> **with A$'$ = { #getReport#y}**
>
> **<x, #getReportτ(String);y> $\cup$ <y, $\bowtie$(#report!(String))>** $\Longrightarrow^{\mathbf{A'}}_{\mathbf{merge\_trivial}}$
>
> **<x, #getReportτ(String);y> $\cup$ <y, #report!(String)>**

We then obtain **B = #getReportτ(String);#report!(String)**.

For a negative case, suppose that we make use of label **#getReport** instead of the label **#report** to transmit the requested report. This would violate the condition of labels being used linearly because then, at one point, we would have two receivers for the empty message sent by the **#forecastWeather** service.

This type of errors are detected by the algorithm when solving the constraints. In our example, the unification algorithm would instead solve variable **y** with type **<y, #getReport!(String)>** which would violate the apartness restriction **#getReport#y** and therefore the unification algorithm would abort. Thus typechecked programs always comply with linear usage of labels inside conversations.

4.3.2 The Calculator Service

As input we have

P = code of the calculator service from the example in 3.2.1
Γ= R = A = ∅

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking site **#operations** we obtain the conversation type **#operations:[#sum](#op1?(Int);#op2?(Int);#res!(Int))**;
3. When typechecking the invocation of service **#sum** via the join primitive, line 14, a type variable **x** is created as well as a constraint to represent the merge of the primitive's body's behaviour with the behaviour of the invoked service, **<x, ⋈(#op1!(x_1);#op2!(x_2), #op1?(Int);#op2?(Int);#res!(Int))>** that is added to R;
4. By typechecking lines 19 to 21, two additional constraints are added **<x_1, Int>** and **<x_2, Int>**;
5. The constraints are resolved upon the typecheck of the site **#utilities** and the algorithm terminates.

The algorithm outputs **(Γ ∪ #utilities:[#calculator](#l1?(Int);#l2?(Int); &{ #div: #res!(Int), #mul: #res!(Int), #sub: #res!(Int), #sum:B }, R, A′)** where **B** is the type obtained by the unification algorithm when solving the constraints on **x**, **x_1**, and **x_2** and **A′** the resulting apartness set.

In step 5 the unification algorithm is called with the following input:

R = { <x, ⋈(#op1!(x_1);#op2!(x_2), #op1?(Int);#op2?(Int);#res!(Int))>,
<x_1, Int>, <x_2, Int> }
A = ∅

So **resolve**(R, A) takes the following steps:

- Starts by solving **<x, ⋈(#op1!(x_1);#op2!(x_2), #op1?(Int);#op2?(Int);#res!(Int))>**

$$\textbf{<x, ⋈(\#op1!(x_1);\#op2!(x_2), \#op1?(Int);\#op2?(Int);\#res!(Int))>} \Longrightarrow^{\textbf{A}}_{\textbf{merge\_sync}}$$

$$\textbf{<x, \#op1}\tau\textbf{(x_1);y)>} \cup \textbf{<y, ⋈(\#op2!(x_2), \#op2?(Int);\#res!(Int))>} \Longrightarrow^{\textbf{A}'}_{\textbf{merge\_sync}}$$
with A′ = { #op1#y }

$$\textbf{<x, \#op1}\tau\textbf{(x_1);y)>} \cup \textbf{<y, \#op2}\tau\textbf{(x_2);z>} \cup \textbf{<z, ⋈(}\emptyset\textbf{, \#res!(Int))>} \Longrightarrow^{\textbf{A}''}_{\textbf{merge\_inact}}$$
with A″ = A′ ∪ { #op2#z }

$$\textbf{<x, \#op1}\tau\textbf{(x_1);y)>} \cup \textbf{<y, \#op2}\tau\textbf{(x_2);z>} \cup \textbf{<z, ⋈(\#res!(Int))>} \Longrightarrow^{\textbf{A}''}_{\textbf{merge\_trivial}}$$

$$\textbf{<x, \#op1}\tau\textbf{(x_1);y)>} \cup \textbf{<y, \#op2}\tau\textbf{(x_2);z>} \cup \textbf{<z, \#res!(Int)>}$$

- Proceeds with **<x$_1$, Int>**

$$\textbf{<x_1, Int>} \Longrightarrow^{\textbf{A}}_{\textbf{merge\_elim}}$$

$$\textbf{<x_1, Int>} \cup \textbf{<x, \#op1}\tau\textbf{(Int);y)>} \cup \textbf{<y, \#op2}\tau\textbf{(x_2);z>} \cup \textbf{<z, \#res!(Int)>}$$

- Terminates by solving **<x$_2$, Int>**

$$\textbf{<x_2, Int>} \Longrightarrow^{\textbf{A}}_{\textbf{merge\_elim}}$$

$$\textbf{<x_2, Int>} \cup \textbf{<x, \#op1}\tau\textbf{(Int);y)>} \cup \textbf{<y, \#op2}\tau\textbf{(Int);z>} \cup \textbf{<z, \#res!(Int)>}$$

We then obtain **B = #op1τ(Int);#op2τ(Int);#res!(Int)**.

Suppose now that the client initialises variable **r** with an empty string. We would then obtain the type **#l1!(Int);#l2!(Int); +{ #div: #res?(String), #mul: #res?(String), #sub: #res?(String), #sum: #res?(String)}** when typechecking the code inside the invoke primitive. But to typecheck an invoke primitive, the type of its body must be dual to invoked service's type, which fails in this case so the typechecking fails altogether. This duality check ensures that a client complies with a protocol usage of the invoked services.

4.3.3 The Online Support Service

As input we have

P = code of the #onlineService site from the example in 3.2.2
Γ = #Support:[#Assistant](#product?(_);#askForProblem!(String);
#problemDetails?(_);#solution!(String);#done!(String))
R = A = ∅

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking the invocation of #Assistant service via **join**, a type variable **x** is created as well as a merge constraint **<x, ⋈(#done?(_);#bye!(String), #product?(_);#askFor–Problem!(String);#problemDetails?(_);#solution!(String);#done!(String))>** that is added to R;
3. The constraint is resolved upon the typecheck of the site **#CompanyHQ** and the algorithm terminates.

The algorithm outputs **(Γ ∪ #CompanyHQ:[#onlineService](#info?(_);&{#commercial:#com–mercial?(_);#product?(_); #details!(String); #bye!(String), #support:#support?(_);B}, R, A′)** where **B** is the type obtained by the unification algorithm when solving the constraint on **x**, and **A′** the resulting apartness set.

In step 3 the unification algorithm is called with the following input:

R = { <x, ⋈(#done?(_);#bye!(String), #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#done!(String))> }
A = ∅

So **resolve**(R, A) takes the following steps:

<x, ⋈(#done?(_);#bye!(String), #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#done!(String))> $\Longrightarrow^{\mathbf{A}}_{\mathbf{merge\_sync}}$

<x, #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#doneτ(String);y)> ∪ <y, ⋈(#bye!(String), ∅)> $\Longrightarrow^{\mathbf{A'}}_{\mathbf{merge\_inact}}$
with A′ = { #done#y}

<x, #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#doneτ(String);y)> ∪ <y, ⋈(#bye!(String))> $\Longrightarrow^{\mathbf{A'}}_{\mathbf{merge\_trivial}}$

<x, #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#doneτ(String);y)> ∪ <y, #bye!(String)>

We then obtain **B = #product?(_); #askForProblem!(String); #problemDetails?(_); #solu–tion!(String);#doneτ(String);#bye!(String)**.

Notice that the ordering and causality of the messages were preserved when we applied the Merge Sync transformation rule. This is because we make a sequential composition of all the behaviour that precedes the synchronised message type (message type with label #done in this example) on each of merged behavioural types and composed them together into a parallel composition of types, *i.e.*:

⋈(#done?(_);#bye!(String), #product?(_); #askForProblem!(String); #problemDetails?(_); #solution!(String);#done!(String))⟶ (∅ | #product?(_);#askForProblem!(String); #problemDetails?(_);#solution!(String));done$_T$(String)

We will now show how the client is typechecked.
As input we have

P = code of the client from the example in 3.2.2
Γ = #Support:[#Assistant](#product?(_);#askForProblem!(String);
#problemDetails?(_);#solution!(String);#done!(String))
R = A = ∅

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking procedure supportDepartment, the identifier supportDepartment is added to Γ with the type **(Bool,_)→$^{\mathbf{B}}$ Unit** where **B = +{#commercial:#commercial!(String); #product!(_);#details?(_);#bye?(_), #support: #support!(String);#product!(_);#askForProblem?(_);#problemDetails!(String);#solution?(_);#bye?(_)}**
3. By typechecking the service invocation at lines 18 and 25, we verify if its body behavioural type complies with the service's behaviour.
4. Upon the typecheck of the parallel composition, a type variable **x** is created as well as a new constraint to represent the merge of all the "visible" behaviour at each invoke, the merge constraint **<x, ⋈(∅, ∅)>** that is added to R.
5. The constraint is resolved at the end of the program.

In step 4 when we say "visible" behaviour of the invokes, we are referring to all the behaviour obtained by the up projection of the invoke's body behavioural type ($\uparrow B$, where B is the invoke's body behavioural type) that is observed as a local behaviour outside the invoke ($loc(\uparrow B)$). Only primitives **sendUp** or **receiveUp** can be observed as having a local behaviour outside the scope of the invoke primitive they were used in.

The only thing noteworthy mentioning on the client's typechecking consists on step 3. In this step we typecheck the first service invocation obtaining its body behavioural type

B$_1$ = #info!(String);+{#commercial: #commercial!(String);#product!(Int);#details?(_); #bye?(_), #support:#support!(String);#product!(Int);#askForProblem?(_);#problemDetails!(String);#solution?(_);#bye?(_)}

and the second service invocation, obtaining the following type for its body

B$_2$ = #info!(String);+{#commercial: #commercial!(String);#product!(String);#details?(_); #bye?(_), #support:#support!(String);#product!(Int);#askForProblem?(_);#problemDetails!(String);#solution?(_);#bye?(_)}

Both these types were obtained by composing together the type of the **send** primitive with the behavioural type of the return type of the procedure call supportDepartment. However the procedure is polymorphic so we obtained different return behavioural types, namely on the type of value sent by label product. These types are dual to the service's type since it is expecting any type on label product.

5 . Implementation

In this chapter we will explain how we implemented the proposed programming language, its distributed runtime support, and typechecking algorithm.

5.1 Tools, Libraries and Design Patterns

We used JavaCC 5 [1] to generate a parser for our language given its grammar. The grammar also contains code to create an abstract syntax tree (AST) when the parser is executed. We then use a visitor design pattern to implement both our evaluation and typechecking algorithms. Thus each AST node has method to accept a visitor algorithm that will visit the node:

```
public void accept(IVisitor v) {
   v.visit(this);
}
```

For the implementation of the distributed runtime support we used Jetty's 6.1 HTTP Server, HTTP Client and libraries [2], included in our **lib** package.

5.2 Language Distributed Runtime Support

Our solution to implement the runtime support follows the architecture presented in Figure 5.1: each machine (rectangle) can have many sites (circles) which in turn can define various services (triangles). Associated with each machine there is a conversation broker (pentagon) that is our runtime system. This conversation broker is a HTTP server and is unique per each program (when a program is executed, a thread with a broker is created and executed until the program ends). For routing purposes, all services have a *signature* representing their protocol of interaction. This signature can either be obtained by pre-processing the code (for the untyped version of our prototype) or by typechecking the program (since types describe these protocols).

Routing To fill the routing maps we compare signature S_1 with signature S_2. If S_1 shares a label l with S_2 and S_1's label has an output polarity, then we route label l to the address of the owner of signature S_2. If S_1 has a label l' and S_2 does not, then if such label occurs again in S_1 we route label l' to the address of the owner of S_1 (this means that it is an internal communication).

Synchronous communication We implemented synchronous communication primitives by associating to each label a Java's synchronous queue. This data structure is suited to implement synchronous channels like those found in process calculi since both its put and get primitives

[1] JavaCC website: https://javacc.dev.java.net, JavaCC Eclipse Plug-in website: http://eclipse-javacc.sourceforge.net/

[2] Jetty website: http://jetty.codehaus.org/jetty/

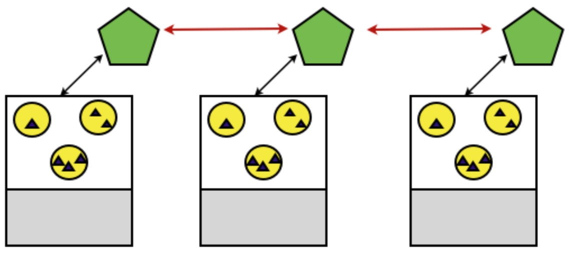


Figure 5.1: Overview of three machines running programs of our language

are blocking, *i.e.* if we try to retrieve a value from the queue it will block until another thread enqueues a value and vice-versa.

We shall now describe further the conversation broker by explaining its actions when evaluating service oriented primitives.

Service definition When evaluating a service definition we create a *closure* with the service's code and associate it with the service's name. This delays the evaluation of the service's definition code until it is invoked. The broker registers the service by associating an AST to the service's name, keeping this information in a structure for this purpose. This AST is a procedure call that will evaluate the service's code upon its invocation. Furthermore, the broker also saves the service's signature in a structure that keeps track of local service's signatures.

Service invocation via invoke Upon service invocation, a HTTP client (referred as client for future reference) is generated by the invoker's broker (hereby denoted as Broker A) to communicate with the service being invoked's broker (from now on denoted as Broker B). Broker A creates a fresh conversation identification by concatenating his address and port number with a counter that is afterwards incremented by one.

The client will send a HTTP request to Broker B and block until his request is handled. The request has a specific header that indicates that it is an invocation request, the freshly generated conversation identification, service to be invoked, the invoker's (Broker A) address and port number, and the invoker's signature (its body signature). When the request is handled, a reply is received by the client. If it states that Broker B is ready to invoke the service, then Broker A fills its routing structure by comparing both his and the received [by reply] service's signature (adding the service's address and signature to a structure, denoted as conversation's partners' structure, that keeps track of each participant in a conversation), and sends a final but non-blocking request (or else the Broker A would get stuck awaiting for the invoked service to end its execution) to say he is ready for the service invocation as well, terminating the client and returning to the evaluation process the conversation identification. The evaluation process makes use of this conversation identification to evaluate the body of the invoker in the correct context (context is therefore the conversation identification used to make requests to the local broker).

On Broker B's end, the initial request for invocation will make Broker B retrieve the service's signature so he can fill his routing table accordingly by comparing with the invoker's labels. Then it will add the address and signature of the invoker to the conversation partner's structure and reply that it is ready to invoke the requested service. When it receives the second request, it will obtain the service's AST and evaluate it under the new conversation identification, ceasing its execution afterwards.

Service invocation via join Asking a service to join a conversation is identical to invoke a service but the evaluation process handles which conversation context the service has by looking into the conversation environment (kept in this process). So on the invoker's side the broker follows the same steps as in a normal service invocation except that it will use a given conversation identification instead of generating one, and, additionally, send routing information to any partner in the conversation (so they can fill their routing maps in case they share a label with the invoked service).

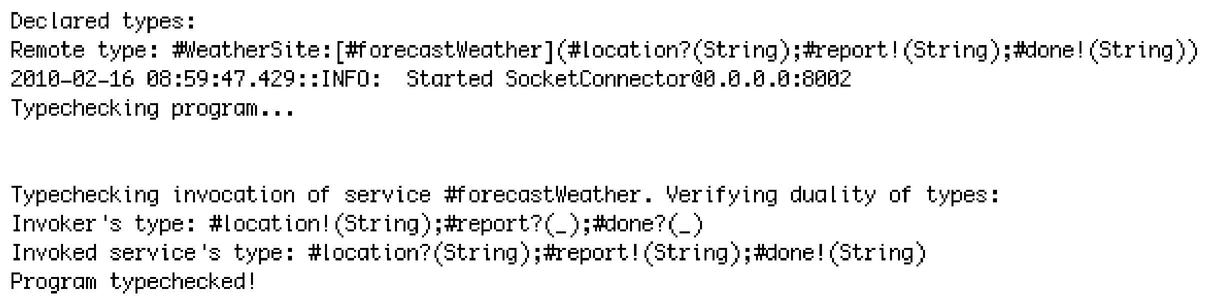
Output An output is evaluated under the current conversation (obtained by looking at the conversation environment) and the evaluation process requests the broker to output a value through a specific label. The broker will make use of the routing information to determine to which participant it must send the value. Then a HTTP client is created to make a request to the destination's broker where the type of the request is sent, the conversation to where it belongs, the *marshalled value*, and a representation of the value's type so it can later be unmarshalled. Notice that this request is blocking so as to ensure a synchronous semantics on communication primitives. On the receiving broker side, it will reconstruct the value received with the information received, then insert the value into the synchronous queue associated with the label and terminate its execution.

Input An input is evaluated under the current conversation, the evaluation process asks the broker to input through a specific label. The broker simply obtains the synchronous queue associated with the label and tries to retrieve a value from it, blocking until it can.

5.3 Typechecking Algorithm

Typechecking. We implemented our typechecking algorithm as a visitor, class **TypeChecking–Visitor**, that transverses the program's AST by "visiting" its nodes (when the node does a callback to the **visit** method of the algorithm). The algorithm is therefore applied recursively, typechecking the program like we would with typing inference rules. During this process it might be necessary to generate and solve constraints. These are added to a set of constraints kept in an object from class **ConstraintSet** (unique throughout the execution of the algorithm). To solve such constraints we use our unification algorithm, implemented as method **resolve** in the **ConstraintSet** class. Figure 5.2 shows the output of our typechecking algorithm when typechecking the code of the client of our weather forecast example 1.3.

The first line shows the declared remote types, then when there is a typecheck of a service invocation via primitive **invoke**, the algorithms outputs the invoked service's type and the body



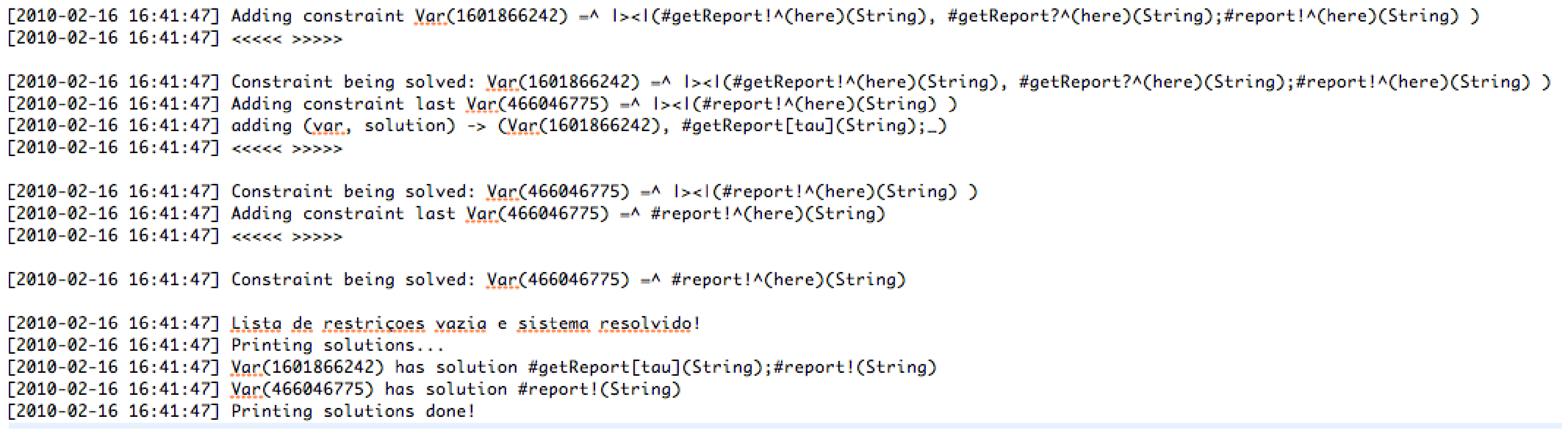
```
Declared types:
Remote type: #WeatherSite:[#forecastWeather](#location?(String);#report!(String);#done!(String))
2010-02-16 08:59:47.429::INFO:  Started SocketConnector@0.0.0.0:8002
Typechecking program...


Typechecking invocation of service #forecastWeather. Verifying duality of types:
Invoker's type: #location!(String);#report?(_);#done?(_)
Invoked service's type: #location?(String);#report!(String);#done!(String)
Program typechecked!
```

Figure 5.2: Output of typechecker

of the invoke primitive's type, so the user can also verify whether or not they're dual. Lastly, the output shows whether the algorithm typechecked the program successfully or if a type error was found.

Type Inference. As we stated above, the type inference algorithm is implemented as a method of class **ConstraintSet**. This class and all the required classes to represent terms of our equational system, are located in the **constraints** package. The **ConstraintSet** class keeps information of all the constraints generated by the typechecking algorithm in a Linked List. Then, for each constraint in the list, the **resolve** method tries to match one of the possible cases defined in 4.2.24 and executes the corresponding actions to solve the constraint. The method terminates when the list is empty. Figure 5.3 shows a trace of the unification algorithm (shown when we execute the interpreter of our language with the debug flag on). The first line corresponds to when the constraint is added during the typechecking of the **join** primitive. The next line is the begin of the unification algorithm. The variable terms we use in the algorithm have an identification number, **Var(id_number)**. When we apply the first transformation rule to the constraint we generate a new variable term, in this case the variable generated has identification number 466046775. Finally, when all the constraints are solved, the algorithm outputs the list of variables solved and their respective types.



```
[2010-02-16 16:41:47] Adding constraint Var(1601866242) =^ |><|(#getReport!^(here)(String), #getReport?^(here)(String);#report!^(here)(String) )
[2010-02-16 16:41:47] <<<<< >>>>>

[2010-02-16 16:41:47] Constraint being solved: Var(1601866242) =^ |><|(#getReport!^(here)(String), #getReport?^(here)(String);#report!^(here)(String) )
[2010-02-16 16:41:47] Adding constraint last Var(466046775) =^ |><|(#report!^(here)(String) )
[2010-02-16 16:41:47] adding (var, solution) -> (Var(1601866242), #getReport[tau](String);_)
[2010-02-16 16:41:47] <<<<< >>>>>

[2010-02-16 16:41:47] Constraint being solved: Var(466046775) =^ |><|(#report!^(here)(String) )
[2010-02-16 16:41:47] Adding constraint last Var(466046775) =^ #report!^(here)(String)
[2010-02-16 16:41:47] <<<<< >>>>>

[2010-02-16 16:41:47] Constraint being solved: Var(466046775) =^ #report!^(here)(String)

[2010-02-16 16:41:47] Lista de restricoes vazia e sistema resolvido!
[2010-02-16 16:41:47] Printing solutions...
[2010-02-16 16:41:47] Var(1601866242) has solution #getReport[tau](String);#report!(String)
[2010-02-16 16:41:47] Var(466046775) has solution #report!(String)
[2010-02-16 16:41:47] Printing solutions done!
```

Figure 5.3: Output of the constraint solving algorithm

6. Concluding Remarks

We have proposed, as this dissertation's main goals, to design and implement a typing and subtyping algorithm with type inference for conversation types. However, these objectives ended up being too ambitious given the time available and the amount of work required to fulfil them. Thus this dissertation's main contribution consists in the conception and implementation of a typing algorithm, with type inference, for conversation types without recursive types and without subtyping.

For that end, we have designed and implemented a programming language for service-oriented computing based on CC (conversation based interactions) which models service primitives through conversations (an abstraction to structured interactions between two or more partners). The language is intended for use in a distributed setting where we can have different sites that provide services that can be invoked locally or remotely. To be able to implement such a distributed language that uses the notion of conversations, we have implemented a distributed runtime support. Since a conversation is a recent abstraction that had no concretization so far, and therefore it was not clear how a conversation could be mapped to existing middleware, its implementation was a challenge itself. This is specially so because the distributed runtime support for our language has to be able to deal with conversations in a distributed environment, which also made its implementation challenging. Therefore, one of the novelties of this work consists in the design and implementation of a prototype proof-of-concept programming language, based on CC, with distributed runtime support. We also believe this programming language to be the first tool to implement multiparty conversations and its respective type system.

We have presented a type system for our language that while it is an adaptation of CC's it has a new operator on behavioural types, the sequential composition. This composition is a necessary addition to represent the sequential behaviour that may come from sequential code that otherwise would be hard to represent through a behavioural type. Also we define what an error program is in our language: a program that at any point has more than a sender or receiver on the same label within the same conversation.

We formalised and implemented a typechecking algorithm with type inference for the proposed language. To do so, we defined a simplification of CC's merge relation on behavioural types. We reduced the type inference problem to a constraint based typing, i.e. constraint generation/constraint solving problem. We represent our constraints on types through a system of equations that is solved through the successive application of transformation rules until the system is either solved or can no longer apply such rules (meaning the algorithm aborts). The solution obtained is thus a mapping of type variables to their respective types. We have introduced new transformation rules besides the standard ones to represent the merge relation we have on behavioural types. These new rules ensure that a merge composition on two behavioural types is such that no unnecessary interleaves are made and that message ordering is kept. This allows us to obtain a principal typing when resolving constraints in our language.

Regarding the validation of this dissertation's work, we have presented and proved the main theoretical results of this dissertation's work: the soundness, completeness, and decidability of our typechecking algorithm. We also prove that we always obtain a principal typing through the application of our typechecking algorithm. Lastly, we tested our prototype's language and typechecking algorithm through examples, some of which are presented in this document.

So this dissertation's contributions are the conception and implementation of a type inference algorithm for conversation types; design and implementation of a prototype for a *proof-of-concept* programming language based on CC; implementation of the language's respective distributed runtime support; the algorithm's completeness, soundness, and decidability proofs along with the proof that the algorithm always returns a principal typing; some validating examples for experimental testing of said algorithm; and the publication of a paper [23] in the pre-proceedings of Programming Language Approaches to Concurrency and Communication-cEntric Software 2010 (PLACES'10) containing some of this work's results.

6.1 Future Work

Due to time limitations, we were unable to conclude the initial work plan and fulfil all the initial objectives of this work. Namely:

Recursive Types. The incorporation of recursive types in our language requires some care. We hint that we would need to take a similar approach to CC's: to divide the labels occurring in a program into two subsets, shared (labels inside a recursion) and plain set of labels. This would enable us to guarantee that a recursive piece of a program could be safely composed together with concurrent instances of that piece as long as those pieces do not interfere with each other and that both the recursive piece and the concurrent instances of it only had labels from the shared set. With respect to the merge relation, we would need to incorporate recursive types in it such that the merge of a recursive behavioural type B with a behavioural type B′ results in the behavioural type $\mathbf{rec}.\mathcal{X}.\mathrm{B}.\mathcal{X} \mid \bowtie(\mathrm{B}, \mathrm{B}')$, where $\mathcal{X}$ is a recursive variable. This implies that the behaviour inside a recursion is persistent. We would have to accommodate this new merge rule in the unification algorithm by adding a corresponding transformation rule. We believe this would not change the theoretical results obtained, namely the decidability result as long as we pay some attention to the implications of the new merge rule. For instance, by the current definition of a term size, we could not prove that the new unification algorithm terminates. To be able to do so we need to redefine term size to include an exception: the size of a parallel composition on terms is the maximum size of all its terms. This is a conservative change since it does not alter the results we obtained for the recursion free theory.

This change suffices to prove decidability because in the worst case we have for some system of equations R, $\mathrm{R} \cup \langle \mathrm{x}, \bowtie(\mathbf{rec}.\mathcal{X}.\mathrm{B}.\mathcal{X}, \mathrm{B}')\rangle \longrightarrow \mathrm{R} \cup \langle \mathrm{x}, \mathbf{rec}.\mathcal{X}.\mathrm{y}.\mathcal{X}\rangle \cup \langle \mathrm{y}, \bowtie(\mathrm{B}, \mathrm{B}')\rangle$ where there is no synchronisation between B and B′ and thus variable y will correspond to the term $\mathrm{B} \mid \mathrm{B}'$. So by using the unaltered term size definition we would obtain $\mathrm{m} + 2{*}\mathrm{m}_1 + \mathrm{m}_2$ with m as the sum of R's terms' sizes, m_1 as the size of term B and m_2 as the size of term

B'. Which is greater than the sum of all terms' size of the system before the transformation $m + m_1 + m_2$. So if we use the new definition of term sizes we obtain $m' + \max(m_1, m_2)$ after the transformation, which is indeed smaller than $m' + m_1 + m_2$, where m' can be smaller due to the new definition.

Subtyping. The inclusion of subtyping in our type system would add more flexibility to our language, allowing, for *e.g.*, the user to only choose a subset of a select interface of options. Subtyping, in our case, does not alter the theoretical results obtained since we only have subtyping in an internal/external choice and we can reduce it to subtyping of records and variants, respectively.

Improvements of the programming language. The runtime support needs to be able to distinguish a conditional choice with behaviour from a conditional choice without behaviour. A possible solution would be, at the parser level, to create a special AST node when parsing a conditional choice that would keep two distinct AST subtrees, one for a normal conditional choice and another for a switch choice, and then have a mechanism, at evaluation time, to know which AST subtree to evaluate.

Regarding the limitation with the unmarshalling of values at runtime level, we could add more information to the marshalled values so as to allow us to know, for instance, their sizes and thus we could distinguish between a separator "," and a string "," during the unmarhsalling process.

Lastly, the invocation of services defined in the same site could be solved by allowing to separate each service definition within the same site to be declared separately as follows: **site** #A { **def** #s_1 **as** { ... } }; **site** #A { **def** #s_2 **as** { ... } }.

Bibliography

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.

[2] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. Scc: A service centered calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[3] Roberto Bruni, Ivan Lanese, Hernán C. Melgratti, and Emilio Tuosto. Multiparty sessions in soc. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[4] Luís Caires and Hugo Torres Vieira. Conversation types. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2009.

[5] Luís Caires and Hugo Torres Vieira. Conversation types. *Technical Report UNL-DI-01-09, Departamento de Informatica, Universidade Nova de Lisboa*, 2009.

[6] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[7] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[8] Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.

[9] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[10] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006.

[11] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer, 1988.

[12] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theor. Comput. Sci.*, 67(2&3):203–260, 1989.

[13] Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[14] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.

[15] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[16] Kohei Honda. Types for dynamic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[17] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[19] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[20] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. 2003.

[21] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002.

[22] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.

[23] Luísa Lourenço and Luís Caires. Type inference for conversation types. In *PLACES'10, 3rd International Workshop in Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2010.

[24] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[25] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In Doug Lea and Gianluigi Zavattaro, editors, *oordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2008.

[26] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[27] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[28] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[29] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.

[30] Matthias Neubauer and Peter Thiemann. An implementation of session types. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[31] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[32] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[33] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[34] François Pottier. Simplifying subtyping constraints. In *In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, 1996.

[35] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[36] Munindar P. Singh and Michael N. Huhns. *Service-oriented Computing - Semantic, Processes, Agents*. John Wiley & Sons, Ltd, 2005.

[37] Geoffrey Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[38] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[39] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[40] Vasco Thudichum Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2004.

[41] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *Programming*

Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[42] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

A . Selected Proofs

A.1 Lemma 4.2.33

Lemma A.1.1. *The relation between the constraint set used to typecheck a program and the resulting constraint set is the following:*

$$\texttt{typecheck}(\mathtt{P},\Gamma,\mathtt{R},\ \mathtt{A}) = (\mathtt{T},\Gamma',\mathtt{R}',\ \mathtt{A}') \quad \Rightarrow \quad \mathtt{R}' = \sigma^{\mathtt{A}}(\mathtt{R}) \cup \mathtt{N}$$

where σ^A is a unifier of R respecting A and N a constraint set representing the new constraints obtained by typechecking program P.

Proof. By induction on the execution of the typecheck algorithm with program P.

Case Pr ⇒ receive(l) :

We want to prove

$$\texttt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\mathtt{K},\Gamma \cup \{\mathtt{this} : \mathtt{l?}(\mathtt{K})\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Since this is a base case, R is empty so for any σ we have $\sigma(R) = \emptyset$ and since there is no new constraint being added, N is also empty thus $R' = \emptyset$.

Case Pr ⇒ send(l, E) :

We want to prove

$$\texttt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma \cup \{\mathtt{this} : \mathtt{l!}(\beta)\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

By I.H. we have $\mathtt{R}' = \sigma(\mathtt{R}) \cup \mathtt{N}$ by typechecking E.

Case Pr ⇒ receiveUp(l) : Similar to **Case Pr ⇒ receive(l)**.

Case Pr ⇒ sendUp(l, E) : Similar to **Case Pr ⇒ send(l, E)**.

Case Pr ⇒ P$_1$; P$_2$:

We want to prove

$$\texttt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\},\mathtt{R}''',\ \mathtt{A}''') \Rightarrow\ \mathtt{R}''' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Then given

$$\texttt{typecheck}(\mathtt{P}_1,\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}_1\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma_1(\mathtt{R}) \cup \mathtt{N}_1$$

and

$$\texttt{typecheck}(\mathtt{P}_2,\Gamma,\mathtt{R}',\ \mathtt{A}') = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_2\},\mathtt{R}'',\ \mathtt{A}'') \Rightarrow\ \mathtt{R}'' = \sigma_2(\mathtt{R}') \cup \mathtt{N}_2$$

We have $\mathtt{R}'' = \sigma_2(\sigma_1(\mathtt{R}) \cup \mathtt{N}_1) \cup \mathtt{N}_2 \Leftrightarrow \mathtt{R}'' = \sigma_2 \circ \sigma_1(\mathtt{R}) \cup \sigma_2(\mathtt{N}_1) \cup \mathtt{N}_2$.
Therefore $\sigma = \sigma_2 \circ \sigma_1$, $A''' = A''$, and $\mathtt{N} = \sigma_2(\mathtt{N}_1) \cup \mathtt{N}_2$.

Case Pr ⇒ $\mathbf{P_1} \| \ldots \| \mathbf{P_n}$:

We want to prove

$$\mathtt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\textstyle\bigcup_\mathtt{i} \Gamma_\mathtt{i} \cup \{\mathtt{this} : \mathtt{K}\}, \bigcup_\mathtt{i} \mathtt{R}_\mathtt{i},\ \bigcup_\mathtt{i} \mathtt{A}_\mathtt{i}) \Rightarrow\ \mathtt{R}''' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Then given

$$\mathtt{typecheck}(\mathtt{P}_\mathtt{i},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_\mathtt{i} \cup \{\mathtt{this} : \mathtt{B}_\mathtt{i}\},\mathtt{R}_\mathtt{i},\ \mathtt{A}_\mathtt{i}) \Rightarrow\ \mathtt{R}_\mathtt{i} = \sigma_\mathtt{i}(\mathtt{R}) \cup \mathtt{N}_\mathtt{i}$$

We have $\sigma = \sigma_\mathtt{i}$ for any $i \in \{1,\ldots,n\}$ since the most general unifier of a system is unique, and $\mathtt{N} = \bigcup \mathtt{N}_\mathtt{i} \cup \{\mathtt{K} \doteq \mathtt{B}\}$.

Case Pr ⇒ let x = E in {P}

We want to prove

$$\mathtt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\},\mathtt{R}''',\ \mathtt{A}''') \Rightarrow\ \mathtt{R}''' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Then given

$$\mathtt{typecheck}(\mathtt{E},\Gamma,\mathtt{R},\ \mathtt{A}) = (\beta,\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}_1\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma_1(\mathtt{R}) \cup \mathtt{N}_1$$

and

$$\mathtt{typecheck}(\mathtt{P},\Gamma_1 \cup \{\mathtt{x} : \beta\},\mathtt{R}',\ \mathtt{A}') = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_2\},\mathtt{R}'',\ \mathtt{A}'') \Rightarrow\ \mathtt{R}'' = \sigma_2(\mathtt{R}') \cup \mathtt{N}_2$$

We have $\mathtt{R}'' = \sigma_2(\sigma_1(\mathtt{R}) \cup \mathtt{N}_1) \cup \mathtt{N}_2 \Leftrightarrow \mathtt{R}'' = \sigma_2 \circ \sigma_1(\mathtt{R}) \cup \sigma_2(\mathtt{N}_1) \cup \mathtt{N}_2$. Therefore $\sigma = \sigma_2 \circ \sigma_1$ and $\mathtt{N} = \sigma_2(\mathtt{N}_1) \cup \mathtt{N}_2$.

Case Pr ⇒ invoke s in n as {P} :

We want to prove

$$\mathtt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \{\mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B})\},\mathtt{R}'',\ \mathtt{A}'') \Rightarrow\ \mathtt{R}'' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Then given

$$\mathtt{typecheck}(\mathtt{P},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma_1(\mathtt{R}) \cup \mathtt{N}_1$$

We have $R'' = \sigma'(R')$ as the result of calling the unification algorithm. Then we have $\sigma'(\sigma_1(R) \cup N_1) \Leftrightarrow \sigma' \circ \sigma_1(R) \cup \sigma'(N_1)$. Therefore $\sigma = \sigma' \circ \sigma_1$ and $N = \sigma'(N_1)$.

Case Pr ⇒ site n {P} :

We want to prove

$$\mathtt{typecheck}(\mathtt{Pr},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \{\mathtt{n} : \mathtt{B}\},\mathtt{R}'',\ \mathtt{A}'') \Rightarrow\ \mathtt{R}'' = \sigma(\mathtt{R}) \cup \mathtt{N}$$

Then given

$$\mathtt{typecheck}(\mathtt{P},\Gamma,\mathtt{R},\ \mathtt{A}) = (\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}\},\mathtt{R}',\ \mathtt{A}') \Rightarrow\ \mathtt{R}' = \sigma_1(\mathtt{R}) \cup \mathtt{N}_1$$

We have $R'' = \sigma'(R')$ as the result of calling the unification algorithm. Then we have $\sigma'(\sigma_1(R) \cup N_1) \Leftrightarrow \sigma' \circ \sigma_1(R) \cup \sigma'(N_1)$. Therefore $\sigma = \sigma' \circ \sigma_1$ and $N = \sigma'(N_1)$.

Case Pr $\Rightarrow$ def s as {P} :

We want to prove
$$\texttt{typecheck(Pr,}\Gamma\texttt{,R, A)} = (\Gamma_1 \cup \{\texttt{this} : [\texttt{s}](\downarrow \texttt{B});\texttt{loc}(\uparrow \texttt{B})\}, \texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma(\texttt{R}) \cup \texttt{N}$$

Then given
$$\texttt{typecheck(P,}\Gamma\texttt{,R, A)} = (\Gamma_1 \cup \{\texttt{this} : \texttt{B}\}, \texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma_1(\texttt{R}) \cup \texttt{N}_1$$

Immediate from I.H., $\sigma = \sigma_1$ and $N = N_1$.

Case Pr $\Rightarrow$ join s in n as {P} :

We want to prove
$$\texttt{typecheck(Pr,}\Gamma\texttt{,R, A)} = (\Gamma_1 \cup \{\texttt{this} : \texttt{K}\}, \texttt{R}'', \texttt{A}'') \Rightarrow \texttt{R}'' = \sigma(\texttt{R}) \cup \texttt{N}$$

Then given
$$\texttt{typecheck(P,}\Gamma\texttt{,R, A)} = (\Gamma_1 \cup \{\texttt{this} : \texttt{B}\}, \texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma_1(\texttt{R}) \cup \texttt{N}_1$$

We have $K = B \bowtie B_1$, $[s](B_1) \in \Gamma \subset \Gamma_1$. So $\sigma = \sigma_1$ and $N = N_1 \cup \{K \doteq B \bowtie B_1\}$.

Case Pr $\Rightarrow$ if(E) then {E$_1$} else {E$_2$}

We want to prove
$$\texttt{typecheck(Pr,}\Gamma\texttt{,R, A)} = (\Gamma \cup \{\texttt{this} : \oplus\{\texttt{l}_1 : \texttt{l}_1;\texttt{B}_1,\ \texttt{l}_2 : \texttt{l}_2;\texttt{B}_2\}\}, \texttt{R}''', \texttt{A}''') \Rightarrow \texttt{R}''' = \sigma(\texttt{R}) \cup \texttt{N}$$

Then given
$$\texttt{typecheck(E,}\Gamma\texttt{,R, A)} = (\beta, \Gamma, \texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma_1(\texttt{R}) \cup \texttt{N}_1$$
and
$$\texttt{typecheck(E}_1\texttt{,}\Gamma\texttt{,R}'\texttt{, A}') = (\beta', \Gamma_1 \cup \{\texttt{this} : \texttt{l}_1;\texttt{B}_1\}, \texttt{R}'', \texttt{A}'') \Rightarrow \texttt{R}'' = \sigma_2(\texttt{R}') \cup \texttt{N}_2$$
and
$$\texttt{typecheck(E}_2\texttt{,}\Gamma\texttt{,R}''\texttt{, A}'') = (\beta', \Gamma_2 \cup \{\texttt{this} : \texttt{l}_2;\texttt{B}_2\}, \texttt{R}''', \texttt{A}''') \Rightarrow \texttt{R}''' = \sigma_3(\texttt{R}'') \cup \texttt{N}_3$$

We have $\texttt{R}''' = \sigma_3(\sigma_2(\texttt{R}') \cup \texttt{N}_2) \cup \texttt{N}_3 \Leftrightarrow \texttt{R}''' = \sigma_3(\sigma_2(\sigma_1(\texttt{R}) \cup \texttt{N}_1) \cup \texttt{N}_2) \cup \texttt{N}_3$
$\Leftrightarrow \texttt{R}''' = \sigma_3 \circ \sigma_2 \circ \sigma_1(\texttt{R}) \cup \sigma_3(\sigma_2(\texttt{N}_1)) \cup \sigma_3(\texttt{N}_2) \cup \texttt{N}_3$.
Therefore $\sigma = \sigma_3\sigma_2 \circ \sigma_1$ and $\texttt{N} = \sigma_3(\sigma_2(\texttt{N}_1)) \cup \sigma_3(\texttt{N}_2) \cup \texttt{N}_3$.

Case Pr $\Rightarrow$ if(E) then {P$_1$} else {P$_2$}

We want to prove
$$\texttt{typecheck(Pr,}\Gamma\texttt{,R, A)} = (\Gamma' \cup \{\texttt{this} : \oplus\{\texttt{l}_1 : \texttt{l}_1;\texttt{B}_1,\ \texttt{l}_2 : \texttt{l}_2;\texttt{B}_2\}\}, \texttt{R}''', \texttt{A}''') \Rightarrow \texttt{R}''' = \sigma(\texttt{R}) \cup \texttt{N}$$

Then given
$$\texttt{typecheck(E,}\Gamma\texttt{,R, A)} = (\beta, \Gamma, \texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma_1(\texttt{R}) \cup \texttt{N}_1$$

$$\text{and}$$
$$\texttt{typecheck}(\texttt{P}_1,\Gamma,\texttt{R}', \texttt{A}') = (\Gamma_1 \cup \{\texttt{this} : \texttt{l}_1;\texttt{B}_1\},\texttt{R}'', \texttt{A}'') \Rightarrow \texttt{R}'' = \sigma_2(\texttt{R}') \cup \texttt{N}_2$$
$$\text{and}$$
$$\texttt{typecheck}(\texttt{P}_2,\Gamma,\texttt{R}'', \texttt{A}'') = (\Gamma_2 \cup \{\texttt{this} : \texttt{l}_2;\texttt{B}_2\},\texttt{R}''', \texttt{A}''') \Rightarrow \texttt{R}''' = \sigma_3(\texttt{R}'') \cup \texttt{N}_3$$

We have $\Gamma' = \Gamma_1 \cup \Gamma_2$, $\texttt{R}''' = \sigma_3(\sigma_2(\texttt{R}') \cup \texttt{N}_2) \cup \texttt{N}_3 \Leftrightarrow \texttt{R}''' = \sigma_3(\sigma_2(\sigma_1(\texttt{R}) \cup \texttt{N}_1) \cup \texttt{N}_2) \cup \texttt{N}_3$
$\Leftrightarrow \texttt{R}''' = \sigma_3 \circ \sigma_2 \circ \sigma_1(\texttt{R}) \cup \sigma_3(\sigma_2(\texttt{N}_1)) \cup \sigma_3(\texttt{N}_2) \cup \texttt{N}_3$.

Therefore $\sigma = \sigma_3\sigma_2 \circ \sigma_1$ and $\texttt{N} = \sigma_3(\sigma_2(\texttt{N}_1)) \cup \sigma_3(\texttt{N}_2) \cup \texttt{N}_3$.

Case $\mathbf{Pr \Rightarrow select\{l_1 : P_1, \ldots, l_n : P_n\}}$:

$$\text{We want to prove}$$
$$\texttt{typecheck}(\texttt{Pr},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma', \textstyle\bigcup_\texttt{i} \texttt{R}_\texttt{i}, \bigcup_\texttt{i} \texttt{A}_\texttt{i}) \Rightarrow \texttt{R}''' = \sigma(\texttt{R}) \cup \texttt{N}$$

$$\text{Then given}$$
$$\texttt{typecheck}(\texttt{P}_\texttt{i},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma_\texttt{i} \cup \{\texttt{this} : \texttt{B}_\texttt{i}\},\texttt{R}_\texttt{i}, \texttt{A}_\texttt{i}) \Rightarrow \texttt{R}_\texttt{i} = \sigma_\texttt{i}(\texttt{R}) \cup \texttt{N}_\texttt{i}$$

We have $\Gamma' = \bigcup_i \Gamma_i \cup \{this : \&\{l_1 : B_1,\ \ldots,\ l_n : B_n\}\}$, $\sigma = \sigma_\texttt{i}$ for any $i \in \{1, \ldots, n\}$ since the most general unifier of a system is unique, and $\texttt{N} = \bigcup \texttt{N}_\texttt{i} \cup \{\texttt{K} \doteq \texttt{B}\}$.

Case $\mathbf{Pr \Rightarrow switch\{case(E_1)\ do\ l_1 : P_1, \ldots,\ case(E_n)\ do\ l_n : P_n,\ default\ do\ l_i : P_d\}}$:

$$\text{We want to prove}$$
$$\texttt{typecheck}(\texttt{Pr},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma', \textstyle\bigcup_\texttt{i} \texttt{R}_\texttt{i} \cup \texttt{R}_\texttt{d}, \bigcup_\texttt{i} \texttt{A}_\texttt{i} \cup \texttt{A}_\texttt{d}) \Rightarrow \texttt{R}''' = \sigma(\texttt{R}) \cup \texttt{N}$$

$$\text{Then given}$$
$$\texttt{typecheck}(\texttt{E}_\texttt{i},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma,\texttt{R}'_\texttt{i}, \texttt{A}'_\texttt{i}) \Rightarrow \texttt{R}'_\texttt{i} = \sigma'_\texttt{i}(\texttt{R}) \cup \texttt{N}'_\texttt{i}$$
$$\text{and}$$
$$\texttt{typecheck}(\texttt{P}_\texttt{i},\Gamma,\texttt{R}'_\texttt{i}, \texttt{A}'_\texttt{i}) = (\Gamma_\texttt{i} \cup \{\texttt{this} : \texttt{B}_\texttt{i}\},\texttt{R}_\texttt{i}, \texttt{A}_\texttt{i}) \Rightarrow \texttt{R}_\texttt{i} = \sigma_\texttt{i}(\texttt{R}'_\texttt{i}) \cup \texttt{N}_\texttt{i}$$
$$\text{and}$$
$$\texttt{typecheck}(\texttt{P}_\texttt{d},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma_\texttt{d} \cup \{\texttt{this} : \texttt{B}_\texttt{d}\},\texttt{R}_\texttt{d}, \texttt{A}_\texttt{d}) \Rightarrow \texttt{R}_\texttt{d} = \sigma_\texttt{d}(\texttt{R}) \cup \texttt{N}_\texttt{d}$$

We have $\Gamma' = \bigcup_i \Gamma_i \cup \{this : \oplus\{l_1 : B_1,\ \ldots,\ l_m : B_m\}\}$ with $m <= n$.

We also have $R_i = \sigma_i(R'_i) \cup N_i \Leftrightarrow \sigma_i(\sigma'_i(R) \cup N'_i) \Leftrightarrow \sigma_i \circ \sigma'_i(R) \cup \sigma_i(N'_i) \cup N_i$. Since the most general unifier of a system is unique, we can have any of the $\sigma_i \circ \sigma'_i$ as the unifier of R.

Therefore $\sigma = \texttt{sigma}_\texttt{i} \circ \sigma'_\texttt{i}$ for any $i \in \{1, \ldots, n\}$, and $\texttt{N} = \bigcup_\texttt{i} \sigma'_\texttt{i}(\texttt{N}'_\texttt{i}) \cup \texttt{N}_\texttt{i} \cup \texttt{N}_\texttt{d}$.

Case $\mathbf{Pr \Rightarrow funf(arg_1, \ldots, arg_n) = \{P\}}$:

$$\text{We want to prove}$$
$$\texttt{typecheck}(\texttt{Pr},\Gamma,\texttt{R}, \texttt{A}) = (\Gamma',\texttt{R}', \texttt{A}') \Rightarrow \texttt{R}' = \sigma(\texttt{R}) \cup \texttt{N}$$

$$\text{Then given}$$
$$\texttt{typecheck}(\texttt{arg}_\texttt{i},\Gamma,\texttt{R}, \texttt{A}) = (\texttt{K}_\texttt{i},\Gamma,\texttt{R}, \texttt{A}) \Rightarrow \texttt{R} = \texttt{R} \cup \emptyset$$
$$\text{and}$$

$$\texttt{typecheck}(\texttt{P},\Gamma'',\texttt{R},\ \texttt{A}) = (\Gamma''',\texttt{R}',\ \texttt{A}') \Rightarrow\ \texttt{R}' = \sigma_1(\texttt{R}) \cup \texttt{N}_1$$

We have for $i \in \{1,\dots,n\}$, $\Gamma' = \Gamma_2 \cup fun$, $fun = \{f : (K_1,\dots,K_n) \rightarrow^B \beta\}$, $\Gamma''' = \Gamma_2 \cup \{arg_1 : K_1,\dots,arg_n : K_n\} \cup \{this : B\}$, $\Gamma'' = \Gamma \cup \{arg_1 : K_1,\dots,arg_n : K_n\}$, and $\Gamma_2 \subset \Gamma$.

Immediate from I.H., $\sigma = \sigma_1$ and $N = N_1$.

Case $\mathbf{Pr \Rightarrow E(E_1,\dots,E_n)}$:

We want to prove

$$\texttt{typecheck}(\texttt{Pr},\Gamma,\texttt{R},\ \texttt{A}) = (\Gamma \cup \{\texttt{this} : \texttt{B}\}, \textstyle\bigcup_{\texttt{i}} \texttt{R}_{\texttt{i}},\ \bigcup_{\texttt{i}} \texttt{A}_{\texttt{i}}) \Rightarrow\ \texttt{R}' = \sigma(\texttt{R}) \cup \texttt{N}$$

Then given

$$\texttt{typecheck}(\texttt{E},\Gamma,\texttt{R},\ \texttt{A}) = ((\beta_1,\dots,\beta_{\texttt{n}}) \rightarrow^{\texttt{B}} \beta,\Gamma,\texttt{R},\ \texttt{A}) \Rightarrow\ \texttt{R} = \texttt{R} \cup \emptyset$$

and

$$\texttt{typecheck}(\texttt{E}_{\texttt{i}},\Gamma,\texttt{R},\ \texttt{A}) = (\beta_{\texttt{i}},\Gamma,\texttt{R}_{\texttt{i}},\ \texttt{A}_{\texttt{i}}) \Rightarrow\ \texttt{R}_{\texttt{i}} = \sigma_{\texttt{i}}(\texttt{R}) \cup \texttt{N}_{\texttt{i}}$$

Therefore $\sigma = \sigma_{\texttt{i}}$ for any $i \in \{1,\dots,n\}$ since the most general unifier of a system is unique, and $\texttt{N} = \bigcup \texttt{N}_{\texttt{i}}$.

□

A.2 Soundness of Typechecking Algorithm

Theorem A.2.1 (Soundness of Typechecking Algorithm)**.** *Let P be a program, Γ, Γ' type environments , T a type, A a set of apartness constraints, and R a constraint set.*

$$\textit{Assume}\quad \texttt{typecheck}(\texttt{P},\Gamma,\emptyset,\emptyset) = (\texttt{T},\Gamma',\texttt{R},\ \texttt{A})\quad \textit{then}$$
$$\texttt{R}(\Gamma') \vdash \texttt{P} : \texttt{R}(\texttt{T})$$
and there is a subtitution θ such that
$$\theta(\texttt{R}(\Gamma')) \vdash \texttt{P} : \theta(\texttt{R}(\texttt{T}))$$

Proof. By induction on the execution of the typecheck algorithm with program P.

Case $\mathbf{Pr \Rightarrow receive(l)}$:

We want to prove $\texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\texttt{K},\Gamma',\emptyset,\emptyset) \Rightarrow\ \Gamma' \vdash \texttt{Pr} : \texttt{K}$
such that $\exists\theta : \theta(\Gamma') \vdash Pr : \theta(K)$
then

Trivially satisfied since we have $\Gamma' = \Gamma \cup \{this : l?(K)\}$, and there exists a substitution θ that satisfies the $\theta(\Gamma') \vdash P : \theta(K)$, namely $\theta = [^{\beta}/_{K}]$, where β is a basic type.

Case $\mathbf{Pr \Rightarrow send(l, E)}$:

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\Gamma',\emptyset,\emptyset) \Rightarrow \ \Gamma' \vdash \mathtt{Pr}$$
$$\text{such that } \exists\theta : \theta(\Gamma') \vdash Pr$$
$$\text{then given}$$
$$\mathtt{typecheck}(\mathtt{E},\Gamma,\emptyset,\emptyset) = (\beta,\Gamma,\emptyset,\emptyset) \xRightarrow{\mathtt{by \quad I.H.}} \Gamma \vdash \mathtt{E} : \beta$$
$$\text{such that } \exists\gamma : \gamma(R(\Gamma)) \vdash E : \gamma(R(\beta))$$

We have $\Gamma' = \Gamma \cup \{this : l!(\beta)\}$ and $\theta = \gamma$.

Case Pr ⇒ receiveUp(l) : Similar to **Case Pr ⇒ receive(l)** :.

Case Pr ⇒ sendUp(l, E) : Similar to **Case Pr ⇒ send(l, E)** :.

Case Pr ⇒ $\mathbf{P_1;P_2}$:

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\mathtt{Unit},\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow$$
$$\mathtt{R}'(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr}$$

$$\text{Then given}$$
$$\mathtt{typecheck}(\mathtt{P}_1,\Gamma,\emptyset,\emptyset) = (\Gamma'_1,\mathtt{R},\mathtt{A}) \xRightarrow{\mathtt{by \quad I.H.}} \mathtt{R}(\Gamma'_1) \vdash \mathtt{P}_1 \ \wedge \ \exists\gamma_1 : \gamma_1(\mathtt{R}(\Gamma'_1)) \vdash \mathtt{P}_1$$
$$\text{and}$$
$$\mathtt{typecheck}(\mathtt{P}_2,\Gamma,\mathtt{R},\mathtt{A}) = (\Gamma'_2,\mathtt{R}',\mathtt{A}') \xRightarrow{\mathtt{by \quad I.H.}} \mathtt{R}'(\Gamma'_2) \vdash \mathtt{P}_2 \ \wedge \ \exists\gamma_2 : \gamma_2(\mathtt{R}'(\Gamma'_2)) \vdash \mathtt{P}_2$$

We have for $i \in \{1,2\}$, $\Gamma'_i = \Gamma_i \cup \{this : B_i\}$, $\Gamma' = \Gamma_1 \cup \Gamma_2 \cup \{this : B_1;B_2\}$.

• By applying Lemma 4.1.7 we have

$$\mathtt{R}(\Gamma_1),\mathtt{R}(\Gamma_2),\mathtt{R}(\{\mathtt{this} : \mathtt{B}_1\}) \vdash \mathtt{P}_1 \ \wedge \ \exists\gamma_1 : \gamma_1(\mathtt{R}(\Gamma_1),\mathtt{R}(\Gamma_2),\mathtt{R}(\{\mathtt{this} : \mathtt{B}_1\})) \vdash \mathtt{P}_1$$
$$\wedge$$
$$\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\mathtt{R}'(\{\mathtt{this} : \mathtt{B}_2\}) \vdash \mathtt{P}_2 \ \wedge \ \exists\gamma_2 : \gamma_2(\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\mathtt{R}'(\{\mathtt{this} : \mathtt{B}_2\})) \vdash \mathtt{P}_2$$

• Also we know by Lemma 4.2.33 that R' can be applied where R would be expected since $R' = \sigma(R) \cup N$ where N is the set of constraints generated by typechecking P_2:

$$\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\mathtt{R}'(\{\mathtt{this} : \mathtt{B}_1\}) \vdash \mathtt{P}_1 \ \wedge \ \exists\gamma_1 : \gamma_1(\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\mathtt{R}'(\{\mathtt{this} : \mathtt{B}_1\})) \vdash \mathtt{P}_1$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\{\mathtt{this} : \mathtt{R}'(\mathtt{B}_1)\} \vdash \mathtt{P}_1 \ \wedge \ \gamma(\mathtt{R}'(\Gamma_1),\mathtt{R}'(\Gamma_2),\{\mathtt{this} : \mathtt{R}'(\mathtt{B}_1)\}) \vdash \mathtt{P}_1$$

$$\wedge$$

$\mathtt{R'(\Gamma_1),R'(\Gamma_2),\{this : R'(B_2)\} \vdash P_2} \ \wedge\ \gamma(\mathtt{R'(\Gamma_1),R'(\Gamma_2),\{this : R'(B_2)\}}) \vdash \mathtt{P_2}$

• Therefore $\mathtt{R'(\Gamma_1,\Gamma_2,this : B_1;B_2) \vdash Pr} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R'(\Gamma_1,\Gamma_2,this : B_1;B_2)}) \vdash \mathtt{Pr}$

Case $\mathbf{Pr \Rightarrow P_1 \| \ldots \| P_n}$:

$$\text{We want to prove } \mathtt{typecheck(Pr,\Gamma,\emptyset,\emptyset) = (Unit,\Gamma',R',A')} \Rightarrow$$
$$\mathtt{R'(\Gamma') \vdash Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R'(\Gamma')}) \vdash \mathtt{Pr}$$

$$\text{Then given}$$
$$\mathtt{typecheck(P_i,\Gamma,\emptyset,\emptyset) = (\Gamma'_i,R_i,A_i)} \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{R_i(\Gamma'_i) \vdash P_i} \ \wedge\ \exists\gamma_i : \gamma_i(\mathtt{R_i(\Gamma'_i)}) \vdash \mathtt{P_i}$$
$$\text{for } i \in \{1,\ldots,n\}$$

We have $\Gamma'_i = \Gamma_i \cup \{this : B_i\}$, $\Gamma' = \bigcup_i \Gamma_i \cup \{this : K\}$, $K = B_1 \bowtie \ldots \bowtie B_n$, $R' = \bigcup_i R_i \cup \{K \doteq B_1 \bowtie \ldots \bowtie B_n\}$, $A' = \bigcup_i A_i$.

• By applying Lemma 4.1.7 we have

$\mathtt{R_i(\Gamma_1),\ldots,R_i(\Gamma_n),R_i(\{this : B_i\}) \vdash P_i} \ \wedge\ \exists\gamma_i : \gamma_i(\mathtt{R_i(\Gamma_1),\ldots,R_i(\Gamma_n),R_i(\{this : B_i\})}) \vdash \mathtt{P_i})$
for $i \in \{1,\ldots,n\}$

• Also we know by Lemma 4.2.33 that $R_i = N_i$ where N_i is the set of constraints generated by typechecking P_i therefore since type variable K is a fresh type variable generated by typechecking Pr, we can use R' where we would expect any of the R_i:

$\mathtt{R'(\Gamma_1),\ldots,R'(\Gamma_n),R'(\{this : B_i\}) \vdash P_i} \ \wedge\ \exists\gamma_i : \gamma_i(\mathtt{R'(\Gamma_1),\ldots,R'(\Gamma_n),R'(\{this : B_i\})}) \vdash \mathtt{P_i}$
for $i \in \{1,\ldots,n\}$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$\mathtt{R'(\Gamma_1),\ldots,R'(\Gamma_n),\{this : R'(B_i)\} \vdash P_i} \ \wedge\ \exists\gamma : \gamma(\mathtt{R'(\Gamma_1),\ldots,R'(\Gamma_n),\{this : R'(B_i)\}}) \vdash \mathtt{P_i}$
for $i \in \{1,\ldots,n\}$

• Therefore $\mathtt{R'(\Gamma_1,\ldots,\Gamma_n,this : K) \vdash Pr} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R'(\Gamma_1,\ldots,\Gamma_n,this : K)}) \vdash \mathtt{Pr}$

Case $\mathbf{Pr \Rightarrow let\ x = E\ in\ \{P\}}$:

$$\text{We want to prove } \texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\texttt{Unit},\Gamma',\texttt{R}',\texttt{A}') \Rightarrow$$
$$\texttt{R}'(\Gamma') \vdash \texttt{Pr} \quad \wedge \quad \exists\theta : \theta(\texttt{R}'(\Gamma')) \vdash \texttt{Pr}$$

Then given

$$\texttt{typecheck}(\texttt{E},\Gamma,\emptyset,\emptyset) = (\beta,\Gamma_1,\texttt{R},\texttt{A}) \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{R}(\Gamma_1) \vdash \texttt{P}_1 : \texttt{R}(\beta) \ \wedge\ \exists\gamma_1 : \gamma_1(\texttt{R}(\Gamma_1)) \vdash \texttt{P}_1 : \texttt{R}(\beta)$$

and

$$\texttt{typecheck}(\texttt{P},\Gamma'_1,\texttt{R},\texttt{A}) = (\Gamma'_2,\texttt{R}',\texttt{A}') \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{R}'(\Gamma'_2) \vdash \texttt{P}_2 \ \wedge\ \exists\gamma_2 : \gamma_2(\texttt{R}'(\Gamma'_2)) \vdash \texttt{P}_2$$

We have $\Gamma_1 = \Gamma \cup \{this : B_1\}$, $\Gamma'_1 = \Gamma_1 \cup \{x : \beta\}$, $\Gamma'_2 = \Gamma_2 \cup \{x : \beta,\ this : B_2\}$, $\Gamma' = \Gamma_2 \cup \{this : B_1; B_2\}$.

• By applying Lemma 4.1.7 we have

$$\texttt{R}(\Gamma),\texttt{R}(\{\texttt{this} : \texttt{B}_1\}),\texttt{R}(\Gamma_2) \vdash \texttt{P}_1 : \texttt{R}(\beta) \ \wedge\ \exists\gamma_1 : \gamma_1(\texttt{R}(\Gamma),\texttt{R}(\{\texttt{this} : \texttt{B}_1\}),\texttt{R}(\Gamma_2)) \vdash \texttt{P}_1 : \texttt{R}(\beta)$$

• Also we know by Lemma 4.2.33 that R' can be applied where R would be expected since $R' = \sigma(R) \cup N$ where N is the set of constraints generated by typechecking P_2:

$$\texttt{R}'(\Gamma),\{\texttt{this} : \texttt{R}'(\texttt{B}_1)\},\texttt{R}'(\Gamma_2) \vdash \texttt{P}_1 : \texttt{R}'(\beta) \ \wedge \exists\gamma_1 : \gamma_1(\texttt{R}'(\Gamma),\{\texttt{this} : \texttt{R}'(\texttt{B}_1)\},\texttt{R}'(\Gamma_2)) \vdash \texttt{P}_1 : \texttt{R}'(\beta)$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\texttt{R}'(\Gamma),\{\texttt{this} : \texttt{R}'(\texttt{B}_1)\},\texttt{R}'(\Gamma_2) \vdash \texttt{P}_1 : \texttt{R}'(\beta) \ \wedge\ \exists\gamma : \gamma(\texttt{R}'(\Gamma),\{\texttt{this} : \texttt{R}'(\texttt{B}_1)\},\texttt{R}'(\Gamma_2)) \vdash \texttt{P}_1 : \texttt{R}'(\beta)$$
$$\wedge$$
$$\texttt{R}'(\Gamma_2),\{\texttt{x} : \texttt{R}'(\beta),\ \texttt{this} : \texttt{R}'(\texttt{B}_2)\} \vdash \texttt{P}_2 \ \wedge\ \exists\gamma : \gamma(\texttt{R}'(\Gamma_2),\{\texttt{x} : \texttt{R}'(\beta),\ \texttt{this} : \texttt{R}'(\texttt{B}_2)\})) \vdash \texttt{P}_2$$

• Therefore $\texttt{R}'(\Gamma_2,\texttt{this} : \texttt{B}_1;\texttt{B}_2) \vdash \texttt{Pr} \quad \wedge \quad \exists\gamma : \gamma(\texttt{R}'(\Gamma_2,\texttt{this} : \texttt{B}_1;\texttt{B}_2)) \vdash \texttt{Pr}$

Case Pr ⇒ invoke s in n as {P} :

$$\text{We want to prove } \texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\texttt{Unit},\Gamma',\texttt{R}',\texttt{A}') \Rightarrow$$
$$\texttt{R}'(\Gamma') \vdash \texttt{Pr} \quad \wedge \quad \exists\theta : \theta(\texttt{R}'(\Gamma')) \vdash \texttt{Pr}$$

Then given

$$\texttt{typecheck}(\texttt{P},\Gamma,\emptyset,\emptyset) = (\Gamma'_1,\texttt{R},\texttt{A}) \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{R}(\Gamma'_1) \vdash \texttt{P} \ \wedge\ \exists\gamma : \gamma(\texttt{R}(\Gamma'_1)) \vdash \texttt{P}$$

We have $\Gamma_1' = \Gamma_1 \cup \{this : B\}$, $\Gamma' = \Gamma_1 \cup \{this : loc(\uparrow B)\}$, $R' = \sigma(R)$, $n : [s](B_1) \in \Gamma \subset \Gamma_1$, and A' the apartness set generated when solving R.

• We know by Lemma 4.2.33 that R' can be applied where R would be expected (since $R' = \sigma(R)$):

$\mathtt{R'(\Gamma_1),R'(\{this : B\}) \vdash P} \wedge \exists\gamma : \gamma(\mathtt{R'(\Gamma_1),R'(\{this : B\})}) \vdash \mathtt{P}$

• Therefore $\mathtt{R'(\Gamma_1, this : loc(\uparrow B)) \vdash Pr} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R'(\Gamma_1, this : loc(\uparrow B))}) \vdash \mathtt{Pr}$

Case Pr ⇒ site n {P} :

$$\text{We want to prove } \mathtt{typecheck(Pr,\Gamma,\emptyset,\emptyset) = (Unit,\Gamma',R',A')} \Rightarrow$$
$$\mathtt{R'(\Gamma') \vdash Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R'(\Gamma')}) \vdash \mathtt{Pr}$$

$$\text{Then given}$$
$$\mathtt{typecheck(P,\Gamma,\emptyset,\emptyset) = (\Gamma_1',R,A)} \xRightarrow{\text{by I.H.}} \mathtt{R(\Gamma_1') \vdash P} \wedge \exists\gamma : \gamma(\mathtt{R(\Gamma_1')}) \vdash \mathtt{P}$$

We have for $\Gamma_1' = \Gamma_1 \cup \{this : B\}$, $\Gamma' = \Gamma_1 \cup \{n : B\}$, $R' = \sigma(R)$, and A' the apartness set generated when solving R.

• We know by Lemma 4.2.33 that R' can be applied where R would be expected (since $R' = \sigma(R)$):

$\mathtt{R'(\Gamma_1),R'(\{this : B\}) \vdash P} \wedge \exists\gamma : \gamma(\mathtt{R'(\Gamma_1),R'(\{this : B\})}) \vdash \mathtt{P}$

• Therefore $\mathtt{R'(\Gamma_1, n : B) \vdash Pr} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R'(\Gamma_1, n : B)}) \vdash \mathtt{Pr}$

Case Pr ⇒ def s as {P} :

$$\text{We want to prove } \mathtt{typecheck(Pr,\Gamma,\emptyset,\emptyset) = (Unit,\Gamma',R,A)} \Rightarrow$$
$$\mathtt{R(\Gamma') \vdash Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R(\Gamma')}) \vdash \mathtt{Pr}$$

$$\text{Then given}$$
$$\mathtt{typecheck(P,\Gamma,\emptyset,\emptyset) = (\Gamma_1',R,A)} \xRightarrow{\text{by I.H.}} \mathtt{R(\Gamma_1') \vdash P} \wedge \exists\gamma : \gamma(\mathtt{R(\Gamma_1')}) \vdash \mathtt{P}$$

Trivially satisfied since we have for $\Gamma_1' = \Gamma_1 \cup \{this : B\}$, $\Gamma' = \Gamma_1 \cup \{this : [s](\downarrow B); loc(\uparrow B)\}$ and $\theta = \gamma$.

Case Pr ⇒ join s in n as {P} :

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\mathtt{Unit},\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow \\ \mathtt{R}'(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr}$$

$$\text{Then given} \\ \mathtt{typecheck}(\mathtt{P},\Gamma,\emptyset,\emptyset) = (\Gamma'_1,\mathtt{R},\mathtt{A}) \xRightarrow{\mathtt{by\ \ I.H.}} \mathtt{R}(\Gamma'_1) \vdash \mathtt{P} \wedge \exists\gamma : \gamma(\mathtt{R}(\Gamma'_1)) \vdash \mathtt{P}$$

We have for $\Gamma'_1 = \Gamma_1 \cup \{this : B\}$, $\Gamma' = \Gamma_1 \cup \{this : K\}$, $K = B_1 \bowtie B$, $R' = R \cup \{K \doteq B_1 \bowtie B\}$, and $n : [s](B_1) \in \Gamma \subset \Gamma_1$.

• Since R' differs from R by one additional (unsolved) constraint, R' can be applied where R would be expected:

$\mathtt{R}'(\Gamma'_1) \vdash \mathtt{P} \wedge \exists\gamma : \gamma(\mathtt{R}'(\Gamma'_1)) \vdash \mathtt{P}$

• Therefore $\mathtt{R}'(\Gamma_1, \mathtt{this} : \mathtt{K}) \vdash \mathtt{Pr} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R}'(\Gamma_1, \mathtt{this} : \mathtt{K})) \vdash \mathtt{Pr}$

Case Pr ⇒ if(E) then {E$_1$} else {E$_2$} :
Sub Case using the IfE-Behaviour rule

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\beta,\Gamma',\mathtt{R}'',\mathtt{A}') \Rightarrow \\ \mathtt{R}''(\Gamma') \vdash \mathtt{Pr} : \mathtt{R}''(\beta) \quad \wedge \quad \exists\theta : \theta(\mathtt{R}''(\Gamma')) \vdash \mathtt{Pr} : \mathtt{R}''(\beta)$$

$$\text{Then given} \\ \mathtt{typecheck}(\mathtt{E},\Gamma,\emptyset,\emptyset) = (\beta_\mathtt{E},\Gamma,\mathtt{R},\mathtt{A}) \xRightarrow{\mathtt{by\ \ I.H.}} \mathtt{R}(\Gamma) \vdash \mathtt{E} : \mathtt{R}(\beta_\mathtt{E}) \wedge \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R}(\Gamma)) \vdash \mathtt{E} : \gamma_\emptyset(\mathtt{R}(\beta_\mathtt{E})) \\ \text{and} \\ \mathtt{typecheck}(\mathtt{E}_1,\Gamma,\mathtt{R},\mathtt{A}) = (\beta,\Gamma'_1,\mathtt{R}',\mathtt{A}') \xRightarrow{\mathtt{by\ \ I.H.}} \\ \mathtt{R}'(\Gamma'_1) \vdash \mathtt{E}_1 : \mathtt{R}'(\beta) \wedge \exists\gamma_1 : \gamma_1(\mathtt{R}'(\Gamma'_1)) \vdash \mathtt{E}_1 : \gamma_1(\mathtt{R}'(\beta)) \\ \text{and} \\ \mathtt{typecheck}(\mathtt{E}_2,\Gamma,\mathtt{R}',\mathtt{A}') = (\beta,\Gamma'_2,\mathtt{R}'',\mathtt{A}'') \xRightarrow{\mathtt{by\ \ I.H.}} \\ \mathtt{R}''(\Gamma'_2) \vdash \mathtt{E}_2 : \mathtt{R}''(\beta) \wedge \exists\gamma_2 : \gamma_2(\mathtt{R}''(\Gamma'_2)) \vdash \mathtt{E}_2 : \gamma_2(\mathtt{R}''(\beta))$$

We have for $i \in \{1,2\}$, $\Gamma'_i = \Gamma_i \cup \{this : l_i;B_i\}$, $\Gamma' = \Gamma \cup \{this : \oplus\{l_1 : l_1;B_1,\ l_2 : l_2;B_2\}\}$. Also R can contain an additional constraint $K \doteq Bool$ in case β_E is a type variable K.

• We know by Lemma 4.2.33 that R'' can be applied where R and R' would be expected since $R' = \sigma(R) \cup N_1$ and $R'' = \sigma(R') \cup N_2$ where N_1 and N_2 are the set of constraints generated by typechecking E_1 and E_2, respectively:
$\mathtt{R}''(\Gamma) \vdash \mathtt{E} : \mathtt{R}''(\beta_\mathtt{E}) \wedge \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R}''(\Gamma)) \vdash \mathtt{E} : \gamma_\emptyset(\mathtt{R}''(\beta_\mathtt{E})$

$$\wedge$$

$\mathtt{R''(\Gamma'_1) \vdash E_1 : R''(\beta)} \;\wedge\; \exists\gamma_1 : \gamma_1(\mathtt{R''(\Gamma'_1)}) \vdash \mathtt{E_1} : \gamma_1(\mathtt{R''}(\beta))$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\begin{array}{l}
\mathtt{R''(\Gamma) \vdash E : R''(\beta_E)} \;\wedge\; \exists\gamma : \gamma(\mathtt{R''(\Gamma)}) \vdash \mathtt{E} : \gamma(\mathtt{R''(\beta_E)}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\mathtt{R''(\Gamma'_1) \vdash E_1 : R''(\beta)} \;\wedge\; \exists\gamma : \gamma(\mathtt{R''(\Gamma'_1)}) \vdash \mathtt{E_1} : \gamma(\mathtt{R''}(\beta)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\mathtt{R''(\Gamma'_2) \vdash E_2 : R''(\beta)} \;\wedge\; \exists\gamma : \gamma(\mathtt{R''(\Gamma'_2)}) \vdash \mathtt{E_2} : \gamma(\mathtt{R''}(\beta))
\end{array}$$

• Therefore $\mathtt{R''(\Gamma') \vdash Pr : R''(\beta)} \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R''(\Gamma')}) \vdash \mathtt{Pr : R''}(\beta)$

Case Pr ⇒ if(E) then {P$_1$} else {P$_2$} :
Sub Case using the If-Behaviour rule

$$\text{We want to prove } \mathtt{typecheck(Pr,\Gamma,\emptyset,\emptyset) = (\Gamma',R'',A')} \Rightarrow$$
$$\mathtt{R''(\Gamma') \vdash Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R''(\Gamma')}) \vdash \mathtt{Pr}$$

Then given

$$\mathtt{typecheck(E,\Gamma,\emptyset,\emptyset) = (\beta_E,\Gamma,R,A)} \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{R(\Gamma) \vdash E : R(\beta_E)} \;\wedge\; \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R(\Gamma)}) \vdash \mathtt{E} : \gamma_\emptyset(\mathtt{R(\beta_E)})$$
and
$$\mathtt{typecheck(P_1,\Gamma,R,A) = (\Gamma'_1,R',A')} \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{R'(\Gamma'_1) \vdash P_1} \;\wedge\; \exists\gamma_1 : \gamma_1(\mathtt{R'(\Gamma'_1)}) \vdash \mathtt{P_1}$$
and
$$\mathtt{typecheck(P_2,\Gamma,R',A') = (\Gamma'_2,R'',A'')} \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{R''(\Gamma'_2) \vdash P_2} \;\wedge\; \exists\gamma_2 : \gamma_2(\mathtt{R''(\Gamma'_2)}) \vdash \mathtt{P_2}$$

We have for $i \in \{1,2\}$, $\Gamma'_i = \Gamma_i \cup \{this : l_i; B_i\}$, $\Gamma' = \Gamma_1 \cup \Gamma_2 \cup \{this : \oplus\{l_1 : l_1; B_1,\ l_2 : l_2; B_2\}\}$. Also R can contain an additional constraint $K \doteq Bool$ in case β_E is a type variable K.

• By applying Lemma 4.1.7 we have:

$$\begin{array}{l}
\mathtt{R(\Gamma),R(\Gamma_1),R(\Gamma_2) \vdash E : R(\beta_E)} \;\wedge\; \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R(\Gamma),R(\Gamma_1),R(\Gamma_2)}) \vdash \mathtt{E} : \gamma_\emptyset(\mathtt{R(\beta_E)}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\mathtt{R'(\Gamma'_1),R'(\Gamma_2) \vdash P_1} \;\wedge\; \exists\gamma_1 : \gamma_1(\mathtt{R'(\Gamma'_1),R'(\Gamma_2)}) \vdash \mathtt{P_1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\mathtt{R''(\Gamma_1),R''(\Gamma'_2) \vdash P_2} \;\wedge\; \exists\gamma_2 : \gamma_2(\mathtt{R''(\Gamma_1),R''(\Gamma'_2)}) \vdash \mathtt{P_2}
\end{array}$$

• Also we know by Lemma 4.2.33 that R'' can be applied where R and R' would be expected since $R' = \sigma(R) \cup N_1$ and $R'' = \sigma(R') \cup N_2$ where N_1 and N_2 are the set of constraints generated by typechecking P_1 and P_2, respectively:

$$\mathtt{R}''(\Gamma),\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma_2) \vdash \mathtt{E} : \mathtt{R}''(\beta_{\mathtt{E}}) \ \wedge\ \exists\gamma_{\emptyset} : \gamma_{\emptyset}(\mathtt{R}''(\Gamma),\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma_2)) \vdash \mathtt{E} : \gamma_{\emptyset}(\mathtt{R}''(\beta_{\mathtt{E}}))$$
$$\wedge$$
$$\mathtt{R}''(\Gamma'_1),\mathtt{R}''(\Gamma_2) \vdash \mathtt{P}_1 \ \wedge\ \exists\gamma_1 : \gamma_1(\mathtt{R}''(\Gamma'_1),\mathtt{R}''(\Gamma_2)) \vdash \mathtt{P}_1$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\mathtt{R}''(\Gamma),\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma_2) \vdash \mathtt{E} : \mathtt{R}''(\beta_{\mathtt{E}}) \ \wedge\ \exists\gamma : \gamma(\mathtt{R}''(\Gamma),\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma_2)) \vdash \mathtt{E} : \gamma(\mathtt{R}''(\beta_{\mathtt{E}}))$$
$$\wedge$$
$$\mathtt{R}''(\Gamma'_1),\mathtt{R}''(\Gamma_2) \vdash \mathtt{P}_1 \ \wedge\ \exists\gamma : \gamma(\mathtt{R}''(\Gamma'_1),\mathtt{R}''(\Gamma_2)) \vdash \mathtt{P}_1$$
$$\wedge$$
$$\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma'_2) \vdash \mathtt{P}_2 \ \wedge\ \exists\gamma : \gamma(\mathtt{R}''(\Gamma_1),\mathtt{R}''(\Gamma'_2)) \vdash \mathtt{P}_2$$

• Therefore $\mathtt{R}''(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}''(\Gamma')) \vdash \mathtt{Pr}$

Case Pr ⇒ select{$\mathbf{l_1 : P_1; \ldots; l_n : P_n}$} :

We want to prove $\mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow$
$\mathtt{R}'(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr}$

Then given

$$\mathtt{typecheck}(\mathtt{P_i},\Gamma,\emptyset,\emptyset) = (\Gamma'_{\mathtt{i}},\mathtt{R_i},\mathtt{A_i}) \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{R_i}(\Gamma'_{\mathtt{i}}) \vdash \mathtt{P_i} \ \wedge\ \exists\gamma_{\mathtt{i}} : \gamma_{\mathtt{i}}(\mathtt{R_i}(\Gamma'_{\mathtt{i}})) \vdash \mathtt{P_i}$$
for $i \in \{1,\ldots,n\}$

We have for $i \in \{1,\ldots,n\}$, $\Gamma'_i = \Gamma_i \cup \{this : B_i\}$, $\Gamma' = \bigcup_i \Gamma_i \cup \{this : \&\{l_1 : B_1,\ldots,l_n : B_n\}\}$, $R' = \bigcup_i R_i$, and $A' = \bigcup_i A_i$.

• By applying Lemma 4.1.7 we have

$\mathtt{R_i}(\Gamma_1),\ldots,\mathtt{R_i}(\Gamma_n),\mathtt{R_i}(\{\mathtt{this} : \mathtt{B_i}\}) \vdash \mathtt{P_i} \ \wedge\ \exists\gamma_{\mathtt{i}} : \gamma_{\mathtt{i}}(\mathtt{R_i}(\Gamma_1),\ldots,\mathtt{R_i}(\Gamma_n),\mathtt{R_i}(\{\mathtt{this} : \mathtt{B_i}\})) \vdash \mathtt{P_i}$
for $i \in \{1,\ldots,n\}$

• Also we know by Lemma 4.2.33 that $R_i = N_i$ where N_i is the set of constraints generated by typechecking P_i so we can use R' where we would expect any of the R_i:

$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\}) \vdash \mathtt{P_i} \ \wedge\ \exists\gamma_{\mathtt{i}} : \gamma_{\mathtt{i}}(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\})) \vdash \mathtt{P_i}$
for $i \in \{1,\ldots,n\}$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\{\mathtt{this} : \mathtt{R}'(\mathtt{B_i})\} \vdash \mathtt{P_i} \;\wedge\; \exists\gamma : \gamma(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\{\mathtt{this} : \mathtt{R}'(\mathtt{B_i})\}) \vdash \mathtt{P_i}$
for $i \in \{1,\ldots,n\}$

• Therefore $\mathtt{R}'(\Gamma_1,\ldots,\Gamma_n,\mathtt{this} : \&\{\mathtt{l_1} : \mathtt{B_1},\ldots,\mathtt{l_n} : \mathtt{B_n}\}) \vdash \mathtt{Pr} \quad \wedge$
$\exists\gamma : \gamma(\mathtt{R}'(\Gamma_1,\ldots,\Gamma_n,\mathtt{this} : \&\{\mathtt{l_1} : \mathtt{B_1},\ldots,\mathtt{l_n} : \mathtt{B_n}\})) \vdash \mathtt{Pr}$

Case Pr ⇒ switch{case(E₁) do l₁ : P₁;...;case(Eₙ) do lₙ : Pₙ;default do lᵢ} :

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow$$
$$\mathtt{R}'(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr}$$

Then given

$$\mathtt{typecheck}(\mathtt{E_i},\Gamma,\emptyset,\emptyset) = (\beta_\mathtt{i},\Gamma,\mathtt{R}'_\mathtt{i},\mathtt{A}'_\mathtt{i}) \xRightarrow{\text{by}\quad\text{I.H.}}$$
$$\mathtt{R}'_\mathtt{i}(\Gamma) \vdash \mathtt{E_i} : \mathtt{R}'_\mathtt{i}(\beta_\mathtt{i}) \;\wedge\; \exists\gamma'_\mathtt{i} : \gamma'_\mathtt{i}(\mathtt{R}'_\mathtt{i}(\Gamma)) \vdash \mathtt{E_i} : \gamma'_\mathtt{i}(\mathtt{R}'_\mathtt{i}(\beta_\mathtt{i}))$$
$$\text{and}$$
$$\mathtt{typecheck}(\mathtt{P_i},\Gamma,\mathtt{R}'_\mathtt{i},\mathtt{A}'_\mathtt{i}) = (\Gamma'_\mathtt{i},\mathtt{R_i},\mathtt{A_i}) \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{R_i}(\Gamma'_\mathtt{i}) \vdash \mathtt{P_i} \;\wedge\; \exists\gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R_i}(\Gamma'_\mathtt{i})) \vdash \mathtt{P_i}$$
$$\text{and}$$
$$\mathtt{typecheck}(\mathtt{P_d},\Gamma,\emptyset,\emptyset) = (\Gamma'_\mathtt{d},\mathtt{R_d},\mathtt{A_d}) \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{R_d}(\Gamma'_\mathtt{d}) \vdash \mathtt{P_d} \;\wedge\; \exists\gamma_\mathtt{d} : \gamma_\mathtt{d}(\mathtt{R_d}(\Gamma'_\mathtt{d})) \vdash \mathtt{P_d}$$

We have for $i \in \{1,\ldots,n\}$, $\Gamma'_i = \Gamma_i \cup \{this : B_i\}$, $\Gamma' = \bigcup_i \Gamma_i \cup \Gamma_d \cup \{this : \oplus\{l_1 : B_1,\ldots,l_m : B_m\}\}$ with $m <= n$, $R' = \bigcup_i R_i \cup R_d$, and $A' = \bigcup_i A_i \cup A_d$. For each R'_i we can have one additional constraint $K_i \doteq Bool$ for a fresh type variable K_i.

• By applying Lemma 4.1.7 we have:

$$\mathtt{R}'_\mathtt{i}(\Gamma_1),\ldots,\mathtt{R}'_\mathtt{i}(\Gamma_n),\mathtt{R}'_\mathtt{i}(\Gamma_d) \vdash \mathtt{E_i} : \mathtt{R}'_\mathtt{i}(\beta_\mathtt{i}) \;\wedge$$
$$\exists\gamma'_\mathtt{i} : \gamma'_\mathtt{i}(\mathtt{R}'_\mathtt{i}(\Gamma_1),\ldots,\mathtt{R}'_\mathtt{i}(\Gamma_n),\mathtt{R}'_\mathtt{i}(\Gamma_d)) \vdash \mathtt{E_i} : \gamma'_\mathtt{i}(\mathtt{R}'_\mathtt{i}(\beta_\mathtt{i}))$$

$$\wedge$$

$$\mathtt{R_i}(\Gamma_1),\ldots,\mathtt{R_i}(\Gamma_n),\mathtt{R_i}(\Gamma_d),\mathtt{R_i}(\{\mathtt{this} : \mathtt{B_i}\}) \vdash \mathtt{P_i} \;\wedge$$
$$\exists\gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R_i}(\Gamma_1),\ldots,\mathtt{R_i}(\Gamma_n),\mathtt{R_i}(\Gamma_d),\mathtt{R_i}(\{\mathtt{this} : \mathtt{B_i}\})) \vdash \mathtt{P_i}$$

$$\wedge$$

$$\mathtt{R_d}(\Gamma_1),\ldots,\mathtt{R_d}(\Gamma_n),\mathtt{R_d}(\Gamma'_d) \vdash \mathtt{P_d} \wedge$$
$$\exists\gamma_d : \gamma_d(\mathtt{R_d}(\Gamma_1),\ldots,\mathtt{R_d}(\Gamma_n),\mathtt{R_d}(\Gamma'_d)) \vdash \mathtt{P_d}$$

• Also we know by Lemma 4.2.33 that R' can be applied where R_i and R'_i would be expected since $R_i \subset R'$ and that each R_i is independent from each other (i.e. only contains constraints generated by P_i), so we can apply R' where we would expect R_i. Furthermore we can also apply R' where R_d is expected:

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d) \vdash \mathtt{E_i} : \mathtt{R}'(\beta_i) \wedge$$
$$\exists\gamma'_i : \gamma'_i(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma''_i)) \vdash \mathtt{E_i} : \gamma'_i(\mathtt{R}'(\beta_i))$$

$$\wedge$$

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\}) \vdash \mathtt{P_i} \wedge$$
$$\exists\gamma_i : \gamma_i(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\})) \vdash \mathtt{P_i}$$

$$\wedge$$

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma'_d) \vdash \mathtt{P_d} \wedge$$
$$\exists\gamma_d : \gamma_d(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma'_d)) \vdash \mathtt{P_d}$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together with γ_d, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n) \cap Dom(\gamma_d)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d)) \vdash \mathtt{E_i} : \mathtt{R}'(\beta_i) \wedge$$
$$\exists\gamma : \gamma(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d))) \vdash \mathtt{E_i} : \gamma(\mathtt{R}'(\beta_i))$$

$$\wedge$$

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\}) \vdash \mathtt{P_i} \wedge$$
$$\exists\gamma : \gamma(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma_d),\mathtt{R}'(\{\mathtt{this} : \mathtt{B_i}\})) \vdash \mathtt{P_i}$$

$$\wedge$$

$$\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma'_d) \vdash \mathtt{P_d} \wedge$$
$$\exists\gamma : \gamma(\mathtt{R}'(\Gamma_1),\ldots,\mathtt{R}'(\Gamma_n),\mathtt{R}'(\Gamma'_d)) \vdash \mathtt{P_d}$$

• Therefore $\mathtt{R}'(\Gamma_1,\ldots,\Gamma_n,\Gamma_d,\mathtt{this} : \oplus\{\mathtt{l_1} : \mathtt{B_1},\ldots,\mathtt{l_n} : \mathtt{B_n}\}) \vdash \mathtt{Pr} \quad \wedge$
$\exists\gamma : \gamma(\mathtt{R}'(\Gamma_1,\ldots,\Gamma_n,\Gamma_d,\mathtt{this} : \oplus\{\mathtt{l_1} : \mathtt{B_1},\ldots,\mathtt{l_n} : \mathtt{B_n}\})) \vdash \mathtt{Pr}$.

Case Pr ⇒ fun f($\mathbf{arg_1,\ldots,arg_n}$) = {P} :

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow$$
$$\mathtt{R}'(\Gamma') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr}$$

$$\text{Then given}$$
$$\mathtt{typecheck}(\mathtt{arg_i},\Gamma,\emptyset,\emptyset) = (\mathtt{K_i},\Gamma,\emptyset,\emptyset) \xRightarrow{\text{by} \quad \text{I.H.}} \Gamma \vdash \mathtt{arg_i} : \mathtt{K_i} \ \wedge\ \exists\gamma_\emptyset : \gamma_\emptyset(\Gamma) \vdash \mathtt{arg_i} : \gamma_\emptyset(\mathtt{K_i})$$
$$\text{and}$$
$$\mathtt{typecheck}(\mathtt{P},\Gamma'',\emptyset,\emptyset) = (\beta,\Gamma''',\mathtt{R}',\mathtt{A}') \xRightarrow{\text{by} \quad \text{I.H.}}$$
$$\mathtt{R}'(\Gamma''') \vdash \mathtt{P} : \mathtt{R}'(\beta) \ \wedge\ \exists\gamma_1 : \gamma_1(\mathtt{R}'(\Gamma''')) \vdash \mathtt{P} : \gamma_1(\mathtt{R}'(\beta))$$

We have for $i \in \{1,\ldots,n\}$, $\Gamma' = \Gamma_2 \cup fun$, $fun = \{f : (K_1,\ldots,K_n) \rightarrow^B \beta\}$, $\Gamma''' = \Gamma_2 \cup \{arg_1 : K_1,\ldots,arg_n : K_n\} \cup \{this : B\}$, $\Gamma'' = \Gamma \cup \{arg_1 : K_1,\ldots,arg_n : K_n\}$, and $\Gamma_2 \subset \Gamma$.

• By applying Lemma 4.1.7 we have:

$\Gamma_2 \vdash \mathtt{arg_i} : \mathtt{K_i} \ \wedge\ \exists\gamma_\emptyset : \gamma_\emptyset(\Gamma_2) \vdash \mathtt{arg_i} : \gamma_\emptyset(\mathtt{K_i})$

• Since R' only contains constraints, we can safely use R' to typecheck each arg_i:

$\mathtt{R}'(\Gamma_2) \vdash \mathtt{arg_i} : \mathtt{R}'(\mathtt{K_i}) \ \wedge\ \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R}'(\Gamma_2)) \vdash \mathtt{arg_i} : \gamma_\emptyset(\mathtt{R}'(\mathtt{K_i}))$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_0 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_0)$ then either type $\gamma_1(x)$ or type $\gamma_0(x)$ is chosen, arbitrarily:

$$\begin{array}{l} \mathtt{R}'(\Gamma_2) \vdash \mathtt{arg_i} : \mathtt{R}'(\mathtt{K_i}) \ \wedge\ \exists\gamma : \gamma(\mathtt{R}'(\Gamma_2)) \vdash \mathtt{arg_i} : \gamma(\mathtt{R}'(\mathtt{K_i})) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ \mathtt{R}'(\Gamma''',\mathtt{fun}) \vdash \mathtt{P} : \mathtt{R}'(\beta) \ \wedge\ \exists\gamma : \gamma(\mathtt{R}'(\Gamma''',\mathtt{fun})) \vdash \mathtt{P} : \gamma(\mathtt{R}'(\beta)) \end{array}$$

• Therefore $\mathtt{R}'(\Gamma''') \vdash \mathtt{Pr} \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma''')) \vdash \mathtt{Pr}$.

Case $\mathbf{Pr \Rightarrow E(E_1,\ldots,E_n)}$:

$$\text{We want to prove } \mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset) = (\beta,\Gamma',\mathtt{R}',\mathtt{A}') \Rightarrow$$
$$\mathtt{R}'(\Gamma') \vdash \mathtt{Pr} : \mathtt{R}'(\beta) \quad \wedge \quad \exists\theta : \theta(\mathtt{R}'(\Gamma')) \vdash \mathtt{Pr} : \theta(\mathtt{R}'(\beta))$$

$$\text{Then given}$$
$$\mathtt{typecheck}(\mathtt{E},\Gamma,\emptyset,\emptyset) = (\beta_\mathtt{f},\Gamma,\emptyset,\emptyset) \xRightarrow{\text{by} \quad \text{I.H.}} \Gamma \vdash \mathtt{E} : \beta_\mathtt{f} \ \wedge\ \exists\gamma_\emptyset : \gamma_\emptyset(\Gamma) \vdash \mathtt{E} : \gamma_\emptyset(\beta_\mathtt{f})$$
$$\text{and}$$
$$\mathtt{typecheck}(\mathtt{E_i},\Gamma,\emptyset,\emptyset) = (\beta_\mathtt{i},\Gamma,\mathtt{R_i},\mathtt{A_i}) \xRightarrow{\text{by} \quad \text{I.H.}}$$
$$\mathtt{R_i}(\Gamma) \vdash \mathtt{E_i} : \mathtt{R_i}(\beta_\mathtt{i}) \ \wedge\ \exists\gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R_i}(\Gamma)) \vdash \mathtt{E_i} : \gamma_\mathtt{i}(\mathtt{R_i}(\beta_\mathtt{i}))$$

We have $\Gamma' = \Gamma \cup \{this : B\}, \beta_f = (\beta_1, \ldots, \beta_n) \rightarrow^B \beta, R' = \bigcup_i R_i, A' = \bigcup_i A_i$.

• By Lemma 4.2.33 we have $R_i = N_i$ where N_i is the set of constraints generated by typechecking E_i so we can use R' where we would expect any of the R_i or where no constraint set was expected:

$$\begin{array}{l} \mathtt{R'}(\Gamma) \vdash \mathtt{E} : \mathtt{R'}(\beta_\mathtt{f}) \ \wedge\ \exists\gamma_\emptyset : \gamma_\emptyset(\mathtt{R'}(\Gamma)) \vdash \mathtt{E} : \gamma_\emptyset(\mathtt{R'}(\beta_\mathtt{f})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ \mathtt{R'}(\Gamma) \vdash \mathtt{E_i} : \mathtt{R'}(\beta_\mathtt{i}) \ \wedge\ \exists\gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R'}(\Gamma)) \vdash \mathtt{E_i} : \gamma_\mathtt{i}(\mathtt{R'}(\beta_\mathtt{i})) \end{array}$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together with γ_0, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n) \cap Dom(\gamma_0)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$$\begin{array}{l} \mathtt{R'}(\Gamma) \vdash \mathtt{E} : \mathtt{R'}(\beta_\mathtt{f}) \ \wedge\ \exists\gamma : \gamma(\mathtt{R'}(\Gamma)) \vdash \mathtt{E} : \gamma(\mathtt{R'}(\beta_\mathtt{f})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ \mathtt{R'}(\Gamma) \vdash \mathtt{E_i} : \mathtt{R'}(\beta_\mathtt{i}) \ \wedge\ \exists\gamma : \gamma(\mathtt{R'}(\Gamma)) \vdash \mathtt{E_i} : \gamma(\mathtt{R'}(\beta_\mathtt{i})) \end{array}$$

• Therefore $\mathtt{R'}(\Gamma, \mathtt{this} : \mathtt{B}) \vdash \mathtt{Pr} : \mathtt{R'}(\beta) \quad \wedge \quad \exists\gamma : \gamma(\mathtt{R'}(\Gamma, \mathtt{this} : \mathtt{B})) \vdash \mathtt{Pr} : \gamma(\mathtt{R'}(\beta))$

□

A.3 Completeness of Typechecking Algorithm

Theorem A.3.1 (Completeness of Typechecking Algorithm). *Let P be a program, Γ a typing context, Γ' a typing context containing only type declarations of remote services, and T a type*

$$\begin{array}{c} \textit{Assume } \Gamma \vdash \mathtt{P} : \mathtt{T} \textit{ then} \\ \mathtt{typecheck}(\mathtt{P}, \Gamma', \emptyset, \emptyset) = (\mathtt{T'}, \Gamma'', \mathtt{R'}, \mathtt{A'}) \\ \textit{and there is a substitution } \theta \textit{ such that} \\ \theta(\mathtt{R'}(\mathtt{T'})) = \mathtt{T} \quad \textit{and} \quad \theta(\mathtt{R'}(\Gamma'')) = \Gamma \quad \textit{and} \quad \theta(\mathtt{R'}(\Gamma')) \subset \Gamma \end{array}$$

Proof. By induction on the structure of P.

Case Pr ⇒ receive(l) :

$$\begin{array}{c} \text{We want to prove } \Gamma, \mathtt{this} : \mathtt{l?}(\beta) \vdash \mathtt{Pr} : \beta \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\mathtt{K}, \Gamma \cup \{\mathtt{this} : \mathtt{l?}(\mathtt{K})\}, \emptyset, \emptyset) \ \wedge \\ \exists\theta : \theta(\mathtt{K}) = \beta \quad \wedge \quad \theta(\Gamma \cup \{\mathtt{this} : \mathtt{l?}(\mathtt{K})\}) = \Gamma \cup \{\mathtt{this} : \mathtt{l?}(\beta)\} \quad \wedge \quad \theta(\Gamma) \subset \Gamma \cup \{this : l?(\beta)\} \end{array}$$

All conditions are trivially satisfied with $\theta = [^{\beta}/_{K}]$.

Case $\mathbf{Pr \Rightarrow send(l,E)}$:

$$\text{We want to prove } \Gamma, \mathtt{this} : \mathtt{l!}(\beta) \vdash \mathtt{Pr} \Rightarrow$$
$$\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma \cup \{\mathtt{this} : \mathtt{l!}(\beta')\}, \mathtt{R}, \mathtt{A}) \wedge$$
$$\exists\theta : \theta(\Gamma \cup \{\mathtt{this} : \mathtt{l!}(\beta')\}) = \Gamma \cup \{\mathtt{this} : \mathtt{l!}(\beta)\} \quad \wedge \quad \theta(\Gamma) \subset \Gamma \cup \{\mathit{this} : l!(\beta)\}$$

$$\text{Then given}$$
$$\Gamma \vdash \mathtt{E} : \beta \xRightarrow{\mathtt{by}\quad \mathtt{I.H.}} \mathtt{typecheck}(\mathtt{E}, \Gamma, \emptyset, \emptyset) = (\beta', \Gamma, \mathtt{R}, \mathtt{A})$$
$$\exists\gamma : \gamma(\mathtt{R}(\beta')) = \beta \quad \wedge \quad \gamma(\mathtt{R}(\Gamma)) = \Gamma \quad \wedge \quad \gamma(\mathtt{R}(\Gamma)) \subset \Gamma$$

All conditions are trivially satisfied with $\theta = \gamma$.

Case $\mathbf{Pr \Rightarrow receiveUp(l)}$: Similar to **Case $\mathbf{Pr \Rightarrow receive(l)}$**

Case $\mathbf{Pr \Rightarrow sendUp(l,E)}$: Similar to **Case $\mathbf{Pr \Rightarrow sendUp(l,E)}$**

Case $\mathbf{Pr \Rightarrow P_1;P_2}$:

$$\text{We want to prove } \Gamma' \vdash \mathtt{Pr} \Rightarrow$$
$$\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \wedge$$
$$\exists\theta : \theta(\mathtt{R}'(\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\})) = \Gamma' \quad \wedge$$
$$\theta(\mathtt{R}'(\Gamma)) \subset \Gamma'$$

$$\text{Then given}$$
$$\Gamma_1, \mathtt{this} : \mathtt{B}_1 \vdash \mathtt{P}_1 \xRightarrow{\mathtt{by}\quad \mathtt{I.H.}} \mathtt{typecheck}(\mathtt{P}_1, \Gamma, \emptyset, \emptyset) = (\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A})$$
$$\exists\gamma_1 : \gamma_1(\mathtt{R}(\Gamma_1, \mathtt{this} : \mathtt{B}'_1)) = \Gamma_1, \mathtt{this} : \mathtt{B}_1 \quad \wedge \quad \gamma_1(\mathtt{R}(\Gamma)) \subset \Gamma_1, \mathtt{this} : \mathtt{B}_1$$
$$\text{and}$$
$$\Gamma_2, \mathtt{this} : \mathtt{B}_2 \vdash \mathtt{P}_2 \xRightarrow{\mathtt{by}\quad \mathtt{I.H.}} \mathtt{typecheck}(\mathtt{P}_2, \Gamma, \mathtt{R}, \mathtt{A}) = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}')$$
$$\exists\gamma_2 : \gamma_2(\mathtt{R}'(\Gamma_2, \mathtt{this} : \mathtt{B}'_2)) = \Gamma_2, \mathtt{this} : \mathtt{B}_2 \quad \wedge \quad \gamma_2(\mathtt{R}'(\Gamma)) \subset \Gamma_2, \mathtt{this} : \mathtt{B}_2$$

Where $\Gamma' = \Gamma_1, \Gamma_2, \mathit{this} : B_1; B_2$.

- By Lemma 4.1.7 we have:

$$(\Gamma_1 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \quad \wedge$$
$$\exists\gamma_1 : \gamma_1(\mathtt{R}(\Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}'_1)) = \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1 \quad \wedge \quad \gamma_1(\mathtt{R}(\Gamma)) \subset \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1$$
$$\text{and}$$

$(\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \Leftrightarrow (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \quad \wedge$
$\exists \gamma_2 : \gamma_2(\mathtt{R}'(\Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}'_2)) = \Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}_2 \quad \wedge \quad \gamma_2(\mathtt{R}'(\Gamma)) \subset \Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}_2$

• Due to Lemma 4.2.33, we know that $R' = \sigma(R) \cup N$, where σ is the unifier of R and N is the set of constraints obtained by typechecking P_2. We then can use R' where R would be expected;

$(\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}', \mathtt{A}') \quad \wedge$
$\exists \gamma_1 : \gamma_1(\mathtt{R}'(\Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}'_1)) = \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1 \quad \wedge \quad \gamma_1(\mathtt{R}'(\Gamma)) \subset \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\begin{gathered}\exists \gamma : \gamma(\mathtt{R}'(\Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}'_1)) = \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1 \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma_2, \Gamma_1, \mathtt{this} : \mathtt{B}_1 \\ \wedge \\ \exists \gamma : \gamma(\mathtt{R}'(\Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}'_2)) = \Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}_2 \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma_1, \Gamma_2, \mathtt{this} : \mathtt{B}_2\end{gathered}$$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma : \gamma(\mathtt{R}'(\Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\})) = \Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\} \quad \wedge$
$\gamma(\mathtt{R}'(\Gamma)) \subset \Gamma_1 \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\}$

Case Pr ⇒ let x = E in {P} :

$$\begin{gathered}\text{We want to prove } \Gamma_2, \mathtt{this} : \mathtt{B}_1;\mathtt{B}_2 \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \wedge \\ \exists \theta : \theta(\mathtt{R}'(\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1;\mathtt{B}'_2\})) = \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\} \quad \wedge \\ \theta(\mathtt{R}'(\Gamma)) \subset \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1;\mathtt{B}_2\}\end{gathered}$$

$$\begin{gathered}\text{Then given} \\ \Gamma, \mathtt{this} : \mathtt{B}_1 \vdash \mathtt{E} : \beta \xRightarrow{\text{by I.H.}} \mathtt{typecheck}(\mathtt{E}, \Gamma, \emptyset, \emptyset) = (\beta', \Gamma \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \\ \exists \gamma_1 : \gamma_1(\mathtt{R}(\beta')) = \beta \quad \wedge \quad \gamma_1(\mathtt{R}(\Gamma, \mathtt{this} : \mathtt{B}'_1)) = \Gamma, \mathtt{this} : \mathtt{B}_1 \quad \wedge \quad \gamma_1(\mathtt{R}(\Gamma)) \subset \Gamma, \mathtt{this} : \mathtt{B}_1 \\ \text{and} \\ \Gamma_2, \mathtt{this} : \mathtt{B}_2, \mathtt{x} : \beta \vdash \mathtt{P} \xRightarrow{\text{by I.H.}} \mathtt{typecheck}(\mathtt{P}, \Gamma \cup \{\mathtt{x} : \beta'\}, \mathtt{R}, \mathtt{A}) = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_2, \mathtt{x} : \beta'\}, \mathtt{R}', \mathtt{A}') \\ \exists \gamma_2 : \gamma_2(\mathtt{R}'(\Gamma_2, \mathtt{this} : \mathtt{B}'_2, \mathtt{x} : \beta')) = \Gamma_2, \mathtt{this} : \mathtt{B}_2, \mathtt{x} : \beta \quad \wedge \quad \gamma_2(\mathtt{R}'(\Gamma)) \subset \Gamma_2, \mathtt{this} : \mathtt{B}_2, \mathtt{x} : \beta\end{gathered}$$

• By Lemma 4.1.7 we have:

$(\beta', \Gamma \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\beta', \Gamma_2 \cup \Gamma \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \wedge$
$\exists \gamma_1 : \gamma_1(\mathtt{R}(\beta')) = \beta \wedge \gamma_1(\mathtt{R}(\Gamma_2, \Gamma, \mathtt{this} : \mathtt{B}'_1)) = \Gamma_2, \Gamma, \mathtt{this} : \mathtt{B}_1 \wedge \gamma_1(\mathtt{R}(\Gamma)) \subset \Gamma_2, \Gamma, \mathtt{this} : \mathtt{B}_1$

• Due to Lemma 4.2.33, we know that $R' = \sigma(R) \cup N$, where σ is the unifier of R and N is the set of constraints obtained by typechecking P. We then can use R' where R would be expected;

$(\beta', \Gamma \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\beta', \Gamma \cup \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma_1 : \gamma_1(\mathtt{R}'(\beta')) = \beta \;\wedge\; \gamma_1(\mathtt{R}'(\Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}'_1)) = \Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}_1 \;\wedge\; \gamma_1(\mathtt{R}'(\Gamma)) \subset \Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}_1$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_1 and γ_2 together, where if $x \in Dom(\gamma_1) \cap Dom(\gamma_2)$ then either type $\gamma_1(x)$ or type $\gamma_2(x)$ is chosen, arbitrarily:

$$\exists \gamma : \gamma(\mathtt{R}'(\beta')) = \beta \;\wedge\; \gamma(\mathtt{R}'(\Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}'_1)) = \Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}_1 \;\wedge\; \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma, \Gamma_2, \mathtt{this} : \mathtt{B}_1$$
$$\wedge$$
$$\exists \gamma : \gamma(\mathtt{R}'(\Gamma_2, \mathtt{this} : \mathtt{B}'_2, \mathtt{x} : \beta')) = \Gamma_2, \mathtt{this} : \mathtt{B}_2, \mathtt{x} : \beta \;\wedge\; \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma_2, \mathtt{this} : \mathtt{B}_2, \mathtt{x} : \beta$$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1 ; \mathtt{B}'_2\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma : \gamma(\mathtt{R}'(\Gamma_2 \cup \{\mathtt{this} : \mathtt{B}'_1 ; \mathtt{B}'_2\})) = \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1 ; \mathtt{B}_2\} \wedge$
$\gamma(\mathtt{R}'(\Gamma)) \subset \Gamma_2 \cup \{\mathtt{this} : \mathtt{B}_1 ; \mathtt{B}_2\}$

Case $\mathbf{Pr \Rightarrow P_1 \| \ldots \| P_n}$ **:**

$$\begin{gathered}\text{We want to prove } \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \ldots \bowtie \mathtt{B}_n \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{K}\}, \mathtt{R}', \mathtt{A}') \wedge \\ \exists \theta : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \mathtt{K}\})) = \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \ldots \bowtie \mathtt{B}_n \quad \wedge \\ \theta(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \ldots \bowtie \mathtt{B}_n\end{gathered}$$

$$\begin{gathered}\text{Then given} \\ \Gamma_i, \mathtt{this} : \mathtt{B}_i \vdash \mathtt{P}_i \xRightarrow{\mathtt{by} \quad \mathtt{I.H.}} \mathtt{typecheck}(\mathtt{P}_i, \Gamma, \emptyset, \emptyset) = (\Gamma_i \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \\ \exists \gamma_i : \gamma_i(\mathtt{R}_i(\Gamma_i, \mathtt{this} : \mathtt{B}'_i)) = \Gamma_i, \mathtt{this} : \mathtt{B}_i \quad \wedge \quad \gamma_i(\mathtt{R}_i(\Gamma)) \subset \Gamma_i, \mathtt{this} : \mathtt{B}_i\end{gathered}$$

With $i \in \{1, \ldots, n\}$, $\Gamma' = \bigcup_i \Gamma_i$, $A' = \bigcup_i A_i$ and $R' = \bigcup_i R_i \cup \{K \doteq B'_1 \bowtie \ldots \bowtie B'_n\}$, where K is a fresh type variable.

• By Lemma 4.1.7 we have:

$(\Gamma_i \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \wedge$
$\exists \gamma_i : \gamma_i(\mathtt{R}_i(\Gamma', \mathtt{this} : \mathtt{B}'_i)) = \Gamma', \mathtt{this} : \mathtt{B}_i \quad \wedge \quad \gamma_i(\mathtt{R}_i(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_i$

• Also we know by Lemma 4.2.33 that $R_i = N_i$ where N_i is the set of constraints generated by typechecking P_i therefore since type variable K is a fresh type variable generated by typechecking Pr, we can use R' where we would expect any of the R_i:

$(\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{i}\}, \mathtt{R_i}, \mathtt{A_i}) \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{i}\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}'_\mathtt{i})) = \Gamma', \mathtt{this} : \mathtt{B_i} \quad \wedge \quad \gamma_\mathtt{i}(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B_i}$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$\exists \gamma : \gamma(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}'_\mathtt{i})) = \Gamma', \mathtt{this} : \mathtt{B_i} \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B_i}$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{K}\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \mathtt{K}\})) = \Gamma', \mathtt{this} : \mathtt{B_1} \bowtie \ldots \bowtie \mathtt{B_n} \quad \wedge$
$\gamma(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B_1} \bowtie \ldots \bowtie \mathtt{B_n}$

Case Pr ⇒ site n {P} :

$$\begin{gathered}\text{We want to prove } \Gamma', \mathtt{n} : \mathtt{B} \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{n} : \mathtt{B}'\}, \mathtt{R}'', \mathtt{A}'') \wedge \\ \exists \theta : \theta(\mathtt{R}''(\Gamma' \cup \{\mathtt{n} : \mathtt{B}'\})) = \Gamma', \mathtt{n} : \mathtt{B} \quad \wedge \\ \theta(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{n} : \mathtt{B}\end{gathered}$$

$$\begin{gathered}\text{Then given} \\ \Gamma', \mathtt{this} : \mathtt{B} \vdash \mathtt{P} \xRightarrow{\mathtt{by} \quad \mathtt{I.H.}} \mathtt{typecheck}(\mathtt{P}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}') \\ \exists \gamma' : \gamma'(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad \wedge \quad \gamma'(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}\end{gathered}$$

• We know by Lemma 4.2.33 that R'' can be applied where R' would be expected (since $R'' = \sigma(R')$):

$(\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}') \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists \gamma' : \gamma'(\mathtt{R}''(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad \wedge \quad \gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{n} : \mathtt{B}'\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists \gamma' : \gamma'(\mathtt{R}''(\Gamma' \cup \{\mathtt{n} : \mathtt{B}'\})) = \Gamma', \mathtt{n} : \mathtt{B} \quad \wedge$
$\gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{n} : \mathtt{B}$

Case Pr ⇒ invoke s in n as{P} :

$$\begin{gathered}\text{We want to prove } \Gamma', \mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}) \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}')\}, \mathtt{R}'', \mathtt{A}'') \wedge \\ \exists \theta : \theta(\mathtt{R}''(\Gamma' \cup \{\mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}')\})) = \Gamma', \mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}) \quad \wedge \\ \theta(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B})\end{gathered}$$

Then given

$$\Gamma', \mathtt{this} : \mathtt{B} \vdash \mathtt{P} \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{typecheck}(\mathtt{P}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}')$$
$$\exists\gamma' : \gamma'(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad\wedge\quad \gamma'(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}$$

With $n : [s](B_1) \in \Gamma$ and $\Gamma \subset \Gamma'$.

• We know by Lemma 4.2.33 that R'' can be applied where R' would be expected (since $R'' = \sigma(R')$):

$(\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}') \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists\gamma' : \gamma'(\mathtt{R}''(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad\wedge\quad \gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}')\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists\gamma' : \gamma'(\mathtt{R}''(\Gamma' \cup \{\mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}')\})) = \Gamma', \mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B}) \quad\wedge$
$\gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma'', \mathtt{this} : \mathtt{loc}(\uparrow \mathtt{B})$

Case Pr ⇒ join s in n as{P} :

We want to prove $\Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \mathtt{B} \vdash \mathtt{Pr} \Rightarrow$
$\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{K}\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists\theta : \theta(\mathtt{R}''(\Gamma' \cup \{\mathtt{this} : \mathtt{K}\})) = \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \mathtt{B} \quad\wedge$
$\theta(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \mathtt{B}$

Then given

$$\Gamma', \mathtt{this} : \mathtt{B} \vdash \mathtt{P} \xRightarrow{\text{by}\quad\text{I.H.}} \mathtt{typecheck}(\mathtt{P}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}')$$
$$\exists\gamma' : \gamma'(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad\wedge\quad \gamma'(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}$$

With $n : [s](B_1) \in \Gamma$ and $\Gamma \subset \Gamma'$, and $R'' = R' \cup \{K \doteq B_1 \bowtie B\}$.

• Since R'' differs from R' by one additional (unsolved) constraint, R'' can be applied where R' would be expected:

$(\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}') \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists\gamma' : \gamma'(\mathtt{R}''(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad\wedge\quad \gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{K}\}, \mathtt{R}'', \mathtt{A}'') \wedge$
$\exists\gamma' : \gamma'(\mathtt{R}''(\Gamma' \cup \{\mathtt{this} : \mathtt{K}\})) = \Gamma', \mathtt{this} : \mathtt{B}_1 \bowtie \mathtt{B} \quad\wedge$
$\gamma'(\mathtt{R}''(\Gamma)) \subset \Gamma'', \mathtt{B}_1 \bowtie \mathtt{B}$

Case Pr ⇒ def s as{P} :

$$\begin{gathered}\text{We want to prove } \Gamma', \mathtt{this} : \mathtt{B}' \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'''\}, \mathtt{R}', \mathtt{A}') \wedge \\ \exists\theta : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \mathtt{B}''\})) = \Gamma', \mathtt{this} : \mathtt{B}''' \quad \wedge \\ \theta(\mathtt{R}''(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}'''\end{gathered}$$

$$\begin{gathered}\text{Then given} \\ \Gamma', \mathtt{this} : \mathtt{B} \vdash \mathtt{P} \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{typecheck}(\mathtt{P}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}') \\ \exists\gamma' : \gamma'(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}')) = \Gamma', \mathtt{this} : \mathtt{B} \quad \wedge \quad \gamma'(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}\end{gathered}$$

With $B''' = [s](\downarrow B);\ loc(\uparrow B)$ and $B'' = [s](\downarrow B');\ loc(\uparrow B')$.

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \mathtt{B}''\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists\gamma' : \gamma'(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \mathtt{B}''\})) = \Gamma', \mathtt{this} : \mathtt{B}''' \quad \wedge$
$\gamma'(\mathtt{R}'(\Gamma)) \subset \Gamma'', \mathtt{this} : \mathtt{B}'''$

Case $\mathbf{Pr} \Rightarrow \mathbf{select}\{\mathbf{l_1} : \mathbf{P_1}, \ldots, \mathbf{l_n} : \mathbf{P_n}\}$:

$$\begin{gathered}\text{We want to prove } \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\} \vdash \mathtt{Pr} \Rightarrow \\ \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\}, \mathtt{R}', \mathtt{A}') \wedge \\ \exists\theta : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\})) = \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\} \quad \wedge \\ \theta(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\}\end{gathered}$$

$$\begin{gathered}\text{Then given} \\ \Gamma_i, \mathtt{this} : \mathtt{B}_i \vdash \mathtt{P}_i \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{typecheck}(\mathtt{P}_i, \Gamma, \emptyset, \emptyset) = (\Gamma_i \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \\ \exists\gamma_i : \gamma_i(\mathtt{R}_i(\Gamma_i, \mathtt{this} : \mathtt{B}'_i)) = \Gamma_i, \mathtt{this} : \mathtt{B}_i \quad \wedge \quad \gamma_i(\mathtt{R}_i(\Gamma)) \subset \Gamma_i, \mathtt{this} : \mathtt{B}_i\end{gathered}$$

With $i \in \{1, \ldots, n\}$, $\Gamma' = \bigcup_i \Gamma_i$, $A' = \bigcup_i A_i$ and $R' = \bigcup_i R_i$.

• By Lemma 4.1.7 we have:

$(\Gamma_i \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \wedge$
$\exists\gamma_i : \gamma_i(\mathtt{R}_i(\Gamma', \mathtt{this} : \mathtt{B}'_i)) = \Gamma', \mathtt{this} : \mathtt{B}_i \quad \wedge \quad \gamma_i(\mathtt{R}_i(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_i$

• Also we know by Lemma 4.2.33 that $R_i = N_i$ where N_i is the set of constraints generated by typechecking P_i therefore we can use R' where we would expect any of the R_i:

$(\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}_i, \mathtt{A}_i) \Leftrightarrow (\Gamma' \cup \{\mathtt{this} : \mathtt{B}'_i\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists\gamma_i : \gamma_i(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}'_i)) = \Gamma', \mathtt{this} : \mathtt{B}_i \quad \wedge \quad \gamma_i(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_i$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$$\exists \gamma : \gamma(\mathtt{R}'(\Gamma', \mathtt{this} : \mathtt{B}'_{\mathtt{i}})) = \Gamma', \mathtt{this} : \mathtt{B}_{\mathtt{i}} \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \mathtt{B}_{\mathtt{i}}$$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\}, \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\})) = \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\} \quad \wedge$
$\gamma(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\}$

Case Pr $\Rightarrow$
switch{case ($\mathbf{E_1}$) do $\mathbf{l_1}$: $\mathbf{P_1}, \ldots$, case ($\mathbf{E_n}$) do $\mathbf{l_n}$: $\mathbf{P_n}$, default do $\mathbf{l_i}$: $\mathbf{P_d}$} :

$$\text{We want to prove } \Gamma', \mathtt{this} : \oplus\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\} \vdash \mathtt{Pr} \Rightarrow$$
$$\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \oplus\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\}, \mathtt{R}', \mathtt{A}') \wedge$$
$$\exists \theta : \theta(\mathtt{R}'(\Gamma' \cup \{\mathtt{this} : \oplus\{\mathtt{l}_1 : \mathtt{B}'_1, \ldots, \mathtt{l}_n : \mathtt{B}'_n\}\})) = \Gamma', \mathtt{this} : \&\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\} \quad \wedge$$
$$\theta(\mathtt{R}'(\Gamma)) \subset \Gamma', \mathtt{this} : \oplus\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_n : \mathtt{B}_n\}$$

$$\text{Then given}$$
$$\Gamma \vdash \mathtt{E}_\mathtt{i} : \beta_\mathtt{i} \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{typecheck}(\mathtt{E}_\mathtt{i}, \Gamma, \emptyset, \emptyset) = (\beta'_\mathtt{i}, \Gamma, \mathtt{R}'_\mathtt{i}, \mathtt{A}'_\mathtt{i})$$
$$\exists \gamma'_\mathtt{i} : \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\beta'_\mathtt{i})) = \beta_\mathtt{i} \quad \wedge \quad \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma)) = \Gamma \quad \wedge \quad \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma)) \subset \Gamma$$
$$\text{and}$$
$$\Gamma_\mathtt{i}, \mathtt{this} : \mathtt{B}_\mathtt{i} \vdash \mathtt{P}_\mathtt{i} \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{typecheck}(\mathtt{P}_\mathtt{i}, \Gamma, \mathtt{R}'_\mathtt{i}, \mathtt{A}'_\mathtt{i}) = (\Gamma_\mathtt{i} \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{i}\}, \mathtt{R}_\mathtt{i}, \mathtt{A}_\mathtt{i})$$
$$\exists \gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma_\mathtt{i}, \mathtt{this} : \mathtt{B}'_\mathtt{i})) = \Gamma_\mathtt{i}, \mathtt{this} : \mathtt{B}_\mathtt{i} \quad \wedge \quad \gamma_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma)) \subset \Gamma_\mathtt{i}, \mathtt{this} : \mathtt{B}_\mathtt{i}$$
$$\text{and}$$
$$\Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}_\mathtt{d} \vdash \mathtt{P}_\mathtt{d} \xRightarrow{\text{by} \quad \text{I.H.}} \mathtt{typecheck}(\mathtt{P}_\mathtt{d}, \Gamma, \emptyset, \emptyset) = (\Gamma_\mathtt{d} \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{d}\}, \mathtt{R}_\mathtt{d}, \mathtt{A}_\mathtt{d})$$
$$\exists \gamma'_\mathtt{d} : \gamma'_\mathtt{d}(\mathtt{R}_\mathtt{d}(\Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}'_\mathtt{d})) = \Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}_\mathtt{d} \quad \wedge \quad \gamma_\mathtt{i}(\mathtt{R}_\mathtt{d}(\Gamma)) \subset \Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}_\mathtt{d}$$

With $i \in \{1, \ldots, n\}$, $m <= n$, $\Gamma' = \bigcup_i \Gamma_i \cup \Gamma_d$, $A' = \bigcup_i A_i \cup A_d$ and $R' = \bigcup_i R_i \cup R_d$ where for each β'_i that is a type variable K_i we have $R'_i = R'_i \cup \{K_i \doteq Bool\}$.

• By Lemma 4.1.7 we have:

$(\beta'_\mathtt{i}, \Gamma, \mathtt{R}'_\mathtt{i}, \mathtt{A}'_\mathtt{i}) \Leftrightarrow (\beta'_\mathtt{i}, \Gamma_1 \cup \ldots \cup \Gamma_\mathtt{n} \cup \Gamma_\mathtt{d}, \mathtt{R}'_\mathtt{i}, \mathtt{A}'_\mathtt{i})$
$\exists \gamma'_\mathtt{i} : \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\beta'_\mathtt{i})) = \beta_\mathtt{i} \quad \wedge \quad \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d})) = \Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d} \wedge \gamma'_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d}$

$$\wedge$$

$(\Gamma_\mathtt{i} \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{i}\}, \mathtt{R}_\mathtt{i}, \mathtt{A}_\mathtt{i}) \Leftrightarrow (\Gamma_1 \cup \ldots \cup \Gamma_\mathtt{n} \cup \Gamma_\mathtt{d} \cup \{\mathtt{this} : \mathtt{B}'_\mathtt{i}\}, \mathtt{R}_\mathtt{i}, \mathtt{A}_\mathtt{i}) \wedge$
$\exists \gamma_\mathtt{i} : \gamma_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}'_\mathtt{i})) = \Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}_\mathtt{i} \wedge$
$\gamma_\mathtt{i}(\mathtt{R}_\mathtt{i}(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_\mathtt{n}, \Gamma_\mathtt{d}, \mathtt{this} : \mathtt{B}_\mathtt{i}$

$$\wedge$$

$(\Gamma_d \cup \{\mathtt{this} : B'_d\}, R_d, A_d) \Leftrightarrow (\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma_d \cup \{\mathtt{this} : B'_d\}, R_d, A_d) \wedge$
$\exists \gamma'_d : \gamma'_d(R_d(\Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B'_d)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d \wedge$
$\gamma'_d(R_d(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d$

• Also we know by Lemma 4.2.33 that $R_i = N_i$ where N_i is the set of constraints generated by typechecking P_i therefore we can use R' where we would expect any of the R_i:

$(\beta'_i, \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma_d, R'_i, A'_i) \Leftrightarrow (\beta'_i, \Gamma_1 \cup \ldots \cup \Gamma_n, \Gamma_d, R', A')$
$\exists \gamma'_i : \gamma'_i(R'(\beta'_i)) = \beta_i \quad \wedge \quad \gamma'_i(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d \wedge$
$\gamma'_i(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d$

$\wedge$

$(\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma_d \cup \{\mathtt{this} : B'_i\}, R_i, A_i) \Leftrightarrow (\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma_d \cup \{\mathtt{this} : B'_i\}, R', A') \wedge$
$\exists \gamma_i : \gamma_i(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B'_i)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_i \wedge$
$\gamma_i(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_i$

$\wedge$

$(\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma_d \cup \{\mathtt{this} : B'_d\}, R_d, A_d) \Leftrightarrow (\Gamma_1 \cup \ldots \cup \Gamma_n, \Gamma_d, \cup\{\mathtt{this} : B'_d\}, R', A') \wedge$
$\exists \gamma'_d : \gamma'_d(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B'_d)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d \wedge$
$\gamma'_d(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all γ_i and γ_d together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n) \cap Dom(\gamma_d)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$\exists \gamma : \gamma(R'(\beta'_i)) = \beta_i \quad \wedge \quad \gamma(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d \wedge$
$\gamma(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d$

$\wedge$

$\exists \gamma : \gamma(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B'_i)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_i \wedge$
$\gamma(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_i$

$\wedge$

$\exists \gamma : \gamma(R'(\Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B'_d)) = \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d \wedge$
$\gamma(R'(\Gamma)) \subset \Gamma_1, \ldots, \Gamma_n, \Gamma_d, \mathtt{this} : B_d$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma' \cup \{\mathtt{this} : \oplus\{l_1 : B'_1, \ldots, l_n : B'_n\}\}, R', A') \wedge$
$\exists \gamma : \gamma(R'(\Gamma' \cup \{\mathtt{this} : \oplus\{l_1 : B'_1, \ldots, l_n : B'_n\}\})) = \Gamma', \mathtt{this} : \oplus\{l_1 : B_1, \ldots, l_n : B_n\} \quad \wedge$
$\gamma(R'(\Gamma)) \subset \Gamma', \mathtt{this} : \oplus\{l_1 : B_1, \ldots, l_n : B_n\}$

Case Pr ⇒ if(E) then {E$_1$} else {E$_2$} :

• Using typing rule (IfE-Behaviour)

We want to prove $\Gamma, \mathtt{this} : \oplus\{l_1 : B_1, l_n : B_n\} \vdash \mathtt{Pr} : \beta \Rightarrow$

$$\texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\beta',\Gamma\cup\{\texttt{this}:\&\{\texttt{l}_1:\texttt{B}'_1,\texttt{l}_n:\texttt{B}'_n\}\},\texttt{R}''',\texttt{A}''') \wedge$$
$$\exists\theta:\theta(\texttt{R}'''(\beta'))=\beta \wedge \theta(\texttt{R}'''(\Gamma\cup\{\texttt{this}:\oplus\{\texttt{l}_1:\texttt{B}'_1,\texttt{l}_n:\texttt{B}'_n\}\})) = \Gamma,\texttt{this}:\oplus\{\texttt{l}_1:\texttt{B}_1,\texttt{l}_n:\texttt{B}_n\} \quad \wedge$$
$$\theta(\texttt{R}'''(\Gamma)) \subset \Gamma,\texttt{this}:\oplus\{\texttt{l}_1:\texttt{B}_1,\texttt{l}_n:\texttt{B}_n\}$$

Then given

$$\Gamma\vdash\texttt{E}:\beta_\emptyset \xRightarrow{\texttt{by}\quad\texttt{I.H.}} \texttt{typecheck}(\texttt{E},\Gamma,\emptyset,\emptyset) = (\beta'_\emptyset,\Gamma,\texttt{R}',\texttt{A}')$$
$$\exists\gamma_\emptyset:\gamma_\emptyset(\texttt{R}'(\beta'_\emptyset))=\beta_\emptyset \quad\wedge\quad \gamma_\emptyset(\texttt{R}'(\Gamma))=\Gamma \quad\wedge\quad \gamma_\emptyset(\texttt{R}'(\Gamma))\subset\Gamma$$

and

$$\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1\vdash\texttt{E}_1:\beta_1 \xRightarrow{\texttt{by}\quad\texttt{I.H.}} \texttt{typecheck}(\texttt{E}_1,\Gamma,\texttt{R}',\texttt{A}') = (\beta'_1,\Gamma\cup\{\texttt{this}:\texttt{l}_1;\texttt{B}'_1\},\texttt{R}'',\texttt{A}'')$$
$$\exists\gamma_1:\gamma_1(\texttt{R}''(\beta'_1))=\beta_1 \wedge \gamma_1(\texttt{R}''(\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}'_1)) = \Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1 \wedge$$
$$\gamma_1(\texttt{R}''(\Gamma))\subset\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1$$

and

$$\Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}_2\vdash\texttt{E}_2:\beta_2 \xRightarrow{\texttt{by}\quad\texttt{I.H.}} \texttt{typecheck}(\texttt{E}_2,\Gamma,\texttt{R}'',\texttt{A}'') = (\beta'_2,\Gamma\cup\{\texttt{this}:\texttt{l}_2;\texttt{B}'_2\},\texttt{R}''',\texttt{A}''')$$
$$\exists\gamma_2:\gamma_2(\texttt{R}'''(\beta'_2))=\beta_2 \wedge \gamma_2(\texttt{R}'''(\Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}'_2)) = \Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}_2 \wedge$$
$$\gamma_2(\texttt{R}'''(\Gamma))\subset\Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}_2$$

If β_E is a type variable K_E then $\{K_E \doteq Bool\} \in R'$.

• We know by Lemma 4.2.33 that $R'' = \sigma(R')\cup N_1$ and $R''' = \sigma(R'')\cup N_2$, where $\sigma(R')$ and $\sigma(R'')$ are the unifiers of R' and R'', respectively, and N_1 and N_2 the set of constraints generated by typechecking E_1 and E_2, respectively. So we can use R''' where we would expect either R' or R':

$$(\beta'_\emptyset,\Gamma,\texttt{R}',\texttt{A}') \Leftrightarrow (\beta'_\emptyset,\Gamma,\texttt{R}''',\texttt{A}''') \wedge$$
$$\exists\gamma_\emptyset:\gamma_\emptyset(\texttt{R}'''(\beta'_\emptyset))=\beta_\emptyset \wedge \gamma_\emptyset(\texttt{R}'''(\Gamma))=\Gamma \wedge \gamma_\emptyset(\texttt{R}'''(\Gamma))\subset\Gamma$$
$$\wedge$$
$$(\beta'_1,\Gamma\cup\{\texttt{this}:\texttt{l}_1;\texttt{B}'_1\},\texttt{R}'',\texttt{A}'') \Leftrightarrow (\beta'_1,\Gamma\cup\{\texttt{this}:\texttt{l}_1;\texttt{B}'_1\},\texttt{R}''',\texttt{A}''')$$
$$\exists\gamma_1:\gamma_1(\texttt{R}'''(\beta'_1))=\beta_1 \wedge \gamma_1(\texttt{R}'''(\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}'_1)) = \Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1 \wedge$$
$$\gamma_1(\texttt{R}'''(\Gamma))\subset\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_0, γ_1 and γ_2 together, where if $x \in Dom(\gamma_0)\cap Dom(\gamma_1)\cap Dom(\gamma_2)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$$\exists\gamma:\gamma(\texttt{R}'''(\beta'_\emptyset))=\beta_\emptyset \wedge \gamma(\texttt{R}'''(\Gamma))=\Gamma \wedge \gamma(\texttt{R}'''(\Gamma))\subset\Gamma$$
$$\wedge$$
$$\exists\gamma:\gamma(\texttt{R}'''(\beta'_1))=\beta_1 \wedge \gamma(\texttt{R}'''(\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}'_1)) = \Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1 \wedge$$
$$\gamma(\texttt{R}'''(\Gamma))\subset\Gamma,\texttt{this}:\texttt{l}_1;\texttt{B}_1$$
$$\wedge$$
$$\exists\gamma:\gamma(\texttt{R}'''(\beta'_2))=\beta_2 \wedge \gamma(\texttt{R}'''(\Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}'_2)) = \Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}_2 \wedge$$
$$\gamma(\texttt{R}'''(\Gamma))\subset\Gamma,\texttt{this}:\texttt{l}_2;\texttt{B}_2$$

• Therefore $\texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\beta',\Gamma \cup \{\texttt{this} : \&\{\texttt{l}_1 : \texttt{B}'_1, \texttt{l}_n : \texttt{B}'_n\}\}, \texttt{R}''', \texttt{A}''') \wedge$ $\exists\gamma : \gamma(\texttt{R}'''(\beta')) = \beta \wedge \gamma(\texttt{R}'''(\Gamma \cup \{\texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}'_1, \texttt{l}_n : \texttt{B}'_n\}\})) = \Gamma, \texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}_1, \texttt{l}_n : \texttt{B}_n\} \wedge$ $\gamma(\texttt{R}'''(\Gamma)) \subset \Gamma, \texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}_1, \texttt{l}_n : \texttt{B}_n\}$

Case Pr ⇒ if(E) then {$\mathbf{P_1}$} else {$\mathbf{P_2}$} :

• Using typing rule (If-Behaviour)

$$\begin{gathered}
\text{We want to prove } \Gamma', \texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}_1, \texttt{l}_n : \texttt{B}_n\} \vdash \texttt{Pr} \Rightarrow \\
\texttt{typecheck}(\texttt{Pr},\Gamma,\emptyset,\emptyset) = (\Gamma' \cup \{\texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}'_1, \texttt{l}_n : \texttt{B}'_n\}\}, \texttt{R}''', \texttt{A}''') \wedge \\
\exists\theta : \theta(\texttt{R}'''(\Gamma' \cup \{\texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}'_1, \texttt{l}_n : \texttt{B}'_n\}\})) = \Gamma', \texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}_1, \texttt{l}_n : \texttt{B}_n\} \quad \wedge \\
\theta(\texttt{R}'''(\Gamma)) \subset \Gamma', \texttt{this} : \oplus\{\texttt{l}_1 : \texttt{B}_1, \texttt{l}_n : \texttt{B}_n\}
\end{gathered}$$

$$\begin{gathered}
\text{Then given} \\
\Gamma \vdash \texttt{E} : \beta_\emptyset \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{typecheck}(\texttt{E},\Gamma,\emptyset,\emptyset) = (\beta'_\emptyset, \Gamma, \texttt{R}', \texttt{A}') \\
\exists\gamma_\emptyset : \gamma_\emptyset(\texttt{R}'(\beta'_\emptyset)) = \beta_\emptyset \quad \wedge \quad \gamma_\emptyset(\texttt{R}'(\Gamma)) = \Gamma \quad \wedge \quad \gamma_\emptyset(\texttt{R}'(\Gamma)) \subset \Gamma \\
\text{and} \\
\Gamma_1, \texttt{this} : \texttt{l}_1;\texttt{B}_1 \vdash \texttt{P}_1 \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{typecheck}(\texttt{P}_1,\Gamma,\texttt{R}',\texttt{A}') = (\Gamma_1 \cup \{\texttt{this} : \texttt{l}_1;\texttt{B}'_1\}, \texttt{R}'', \texttt{A}'') \\
\exists\gamma_1 : \gamma_1(\texttt{R}''(\Gamma_1, \texttt{this} : \texttt{l}_1;\texttt{B}'_1)) = \Gamma_1, \texttt{this} : \texttt{l}_1;\texttt{B}_1 \quad \wedge \quad \gamma_1(\texttt{R}''(\Gamma)) \subset \Gamma_1, \texttt{this} : \texttt{l}_1;\texttt{B}_1 \\
\text{and} \\
\Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}_2 \vdash \texttt{P}_2 \xRightarrow{\texttt{by} \quad \texttt{I.H.}} \texttt{typecheck}(\texttt{P}_2,\Gamma,\texttt{R}'',\texttt{A}'') = (\Gamma_2 \cup \{\texttt{this} : \texttt{l}_2;\texttt{B}'_2\}, \texttt{R}''', \texttt{A}''') \\
\exists\gamma_2 : \gamma_2(\texttt{R}'''(\Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}'_2)) = \Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}_2 \quad \wedge \quad \gamma_2(\texttt{R}'''(\Gamma)) \subset \Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}_2
\end{gathered}$$

With $\Gamma' = \Gamma_1 \cup \Gamma_2$, and if β_E is a type variable K_E then $\{K_E \doteq \mathit{Bool}\} \in R'$.
• By Lemma 4.1.7 we have:

$$\begin{gathered}
(\beta'_\emptyset, \Gamma, \texttt{R}', \texttt{A}') \Leftrightarrow (\beta'_\emptyset, \Gamma_1 \cup \Gamma_2, \texttt{R}', \texttt{A}') \wedge \\
\exists\gamma_\emptyset : \gamma_\emptyset(\texttt{R}'(\beta'_\emptyset)) = \beta_\emptyset \quad \wedge \quad \gamma_\emptyset(\texttt{R}'(\Gamma_1,\Gamma_2)) = \Gamma_1,\Gamma_2 \quad \wedge \quad \gamma_\emptyset(\texttt{R}'(\Gamma)) \subset \Gamma_1,\Gamma_2 \\
\wedge \\
(\Gamma_1 \cup \{\texttt{this} : \texttt{l}_1;\texttt{B}'_1\}, \texttt{R}'', \texttt{A}'') \Leftrightarrow (\Gamma_1 \cup \Gamma_2 \cup \{\texttt{this} : \texttt{l}_1;\texttt{B}'_1\}, \texttt{R}'', \texttt{A}'') \wedge \\
\exists\gamma_1 : \gamma_1(\texttt{R}''(\Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_1;\texttt{B}'_1)) = \Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_1;\texttt{B}_1 \wedge \gamma_1(\texttt{R}''(\Gamma)) \subset \Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_1;\texttt{B}_1 \\
\wedge \\
(\Gamma_2 \cup \{\texttt{this} : \texttt{l}_2;\texttt{B}'_2\}, \texttt{R}''', \texttt{A}''') \Leftrightarrow (\Gamma_1, \Gamma_2 \cup \{\texttt{this} : \texttt{l}_2;\texttt{B}'_2\}, \texttt{R}''', \texttt{A}''') \wedge \\
\exists\gamma_2 : \gamma_2(\texttt{R}'''(\Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}'_2)) = \Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}_2 \wedge \gamma_2(\texttt{R}'''(\Gamma)) \subset \Gamma_1,\Gamma_2, \texttt{this} : \texttt{l}_2;\texttt{B}_2
\end{gathered}$$

• Also we know by Lemma 4.2.33 that $R'' = \sigma(R') \cup N_1$ and $R''' = \sigma(R'') \cup N_2$, where $\sigma(R')$ and $\sigma(R'')$ are the unifiers of R' and R'', respectively, and N_1 and N_2 the set of constraints generated by typechecking P_1 and P_2, respectively. So we can use R''' where we would expect either R' or R':

$$(\beta'_{\emptyset},\Gamma_1\cup\Gamma_2,\mathtt{R}',\mathtt{A}') \Leftrightarrow (\beta'_{\emptyset},\Gamma_1\cup\Gamma_2,\mathtt{R}''',\mathtt{A}''') \wedge$$
$$\exists\gamma_{\emptyset}:\gamma_{\emptyset}(\mathtt{R}'''(\beta'_{\emptyset}))=\beta_{\emptyset} \quad\wedge\quad \gamma_{\emptyset}(\mathtt{R}'''(\Gamma_1,\Gamma_2))=\Gamma_1,\Gamma_2 \quad\wedge\quad \gamma_{\emptyset}(\mathtt{R}'''(\Gamma))\subset\Gamma_1,\Gamma_2$$
$$\wedge$$
$$(\Gamma_1\cup\Gamma_2\cup\{\mathtt{this}:\mathtt{l}_1;\mathtt{B}'_1\},\mathtt{R}'',\mathtt{A}'') \Leftrightarrow (\Gamma_1\cup\Gamma_2\cup\{\mathtt{this}:\mathtt{l}_1;\mathtt{B}'_1\},\mathtt{R}''',\mathtt{A}''') \wedge$$
$$\exists\gamma_1:\gamma_1(\mathtt{R}'''(\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}'_1))=\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}_1 \wedge \gamma_1(\mathtt{R}'''(\Gamma))\subset\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}_1$$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing γ_0, γ_1 and γ_2 together, where if $x\in Dom(\gamma_0)\cap Dom(\gamma_1)\cap Dom(\gamma_2)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$$\exists\gamma:\gamma(\mathtt{R}'''(\beta'_{\emptyset}))=\beta_{\emptyset} \quad\wedge\quad \gamma(\mathtt{R}'''(\Gamma))=\Gamma \quad\wedge\quad \gamma(\mathtt{R}'''(\Gamma))\subset\Gamma$$
$$\wedge$$
$$\exists\gamma:\gamma(\mathtt{R}'''(\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}'_1))=\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}_1 \wedge \gamma(\mathtt{R}'''(\Gamma))\subset\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_1;\mathtt{B}_1$$
$$\wedge$$
$$\exists\gamma:\gamma(\mathtt{R}'''(\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_2;\mathtt{B}'_2))=\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_2;\mathtt{B}_2 \wedge \gamma(\mathtt{R}'''(\Gamma))\subset\Gamma_1,\Gamma_2,\mathtt{this}:\mathtt{l}_2;\mathtt{B}_2$$

• Therefore $\mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset)=(\Gamma'\cup\{\mathtt{this}:\oplus\{\mathtt{l}_1:\mathtt{B}'_1,\mathtt{l}_n:\mathtt{B}'_n\}\},\mathtt{R}''',\mathtt{A}''')\wedge$ $\exists\gamma:\gamma(\mathtt{R}'''(\Gamma'\cup\{\mathtt{this}:\oplus\{\mathtt{l}_1:\mathtt{B}'_1,\mathtt{l}_n:\mathtt{B}'_n\}\}))=\Gamma',\mathtt{this}:\oplus\{\mathtt{l}_1:\mathtt{B}_1,\mathtt{l}_n:\mathtt{B}_n\} \quad\wedge$ $\gamma(\mathtt{R}'''(\Gamma))\subset\Gamma',\mathtt{this}:\oplus\{\mathtt{l}_1:\mathtt{B}_1,\mathtt{l}_n:\mathtt{B}_n\}$

Case Pr ⇒ fun f(arg$_1$,...,arg$_n$) = {P} :

$$\text{We want to prove } \Gamma'''\vdash\mathtt{Pr}\Rightarrow$$
$$\mathtt{typecheck}(\mathtt{Pr},\Gamma,\emptyset,\emptyset)=(\Gamma'',\mathtt{R}',\mathtt{A}')\wedge$$
$$\exists\theta:\theta(\mathtt{R}'(\Gamma''))=\Gamma''' \quad\wedge$$
$$\theta(\mathtt{R}'''(\Gamma))\subset\Gamma'''$$

$$\text{Then given}$$
$$\Gamma\vdash\mathtt{arg_i}:\beta_{\mathtt{i}}\xRightarrow{\text{by I.H.}}\mathtt{typecheck}(\mathtt{arg_i},\Gamma,\emptyset,\emptyset)=(\beta'_{\mathtt{i}},\Gamma,\mathtt{R},\mathtt{A})$$
$$\exists\gamma_{\mathtt{i}}:\gamma_{\mathtt{i}}(\mathtt{R}(\beta'_{\mathtt{i}}))=\beta_{\mathtt{i}} \quad\wedge\quad \gamma_{\mathtt{i}}(\mathtt{R}(\Gamma))=\Gamma \quad\wedge\quad \gamma_{\mathtt{i}}(\mathtt{R}(\Gamma))\subset\Gamma$$
$$\text{and}$$
$$\Gamma_{\emptyset},\mathtt{this}:\mathtt{B}\vdash\mathtt{P}\xRightarrow{\text{by I.H.}}\mathtt{typecheck}(\mathtt{P},\Gamma'_{\emptyset},\mathtt{R},\mathtt{A})=(\Gamma''_{\emptyset}\cup\{\mathtt{this}:\mathtt{B}'\},\mathtt{R}',\mathtt{A}')$$
$$\exists\gamma':\gamma'(\mathtt{R}'(\Gamma''_{\emptyset},\mathtt{this}:\mathtt{B}'))=\Gamma_{\emptyset},\mathtt{this}:\mathtt{B} \quad\wedge\quad \gamma'(\mathtt{R}'(\Gamma))\subset\Gamma_{\emptyset},\mathtt{this}:\mathtt{B}$$

With $i\in\{1,\dots,n\}$, $\Gamma''=\Gamma_2\cup\{f:(K_1,\dots,K_n)\rightarrow^{B'}\beta'\}$, $\Gamma'''=\Gamma_2\cup\{f:(\beta_1,\dots,\beta_n)\rightarrow^{B}\beta\}$, $\Gamma_0=\Gamma_2,arg_1\beta_1,\dots,arg_n:\beta_n$, $\Gamma'_0=\Gamma,arg_1:K_1,\dots,arg_n:K_n$, and $\Gamma''_0=\Gamma_2,arg_1:K_1,\dots,arg_n:K_n$.

• By Lemma 4.1.7 we have:

$(\beta'_{\mathtt{i}}, \Gamma, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\beta'_{\mathtt{i}}, \Gamma_2, \mathtt{R}, \mathtt{A})$
$\exists \gamma_{\mathtt{i}} : \gamma_{\mathtt{i}}(\mathtt{R}(\beta'_{\mathtt{i}})) = \beta_{\mathtt{i}} \quad \wedge \quad \gamma_{\mathtt{i}}(\mathtt{R}(\Gamma_2)) = \Gamma_2 \quad \wedge \quad \gamma_{\mathtt{i}}(\mathtt{R}(\Gamma)) \subset \Gamma_2$

• Also we know by Lemma 4.2.33 that $R' = N$ where N is the set of constraints generated by typechecking P. So we can use R' where we would expect R:

$(\beta'_{\mathtt{i}}, \Gamma_2, \mathtt{R}, \mathtt{A}) \Leftrightarrow (\beta'_{\mathtt{i}}, \Gamma_2, \mathtt{R}', \mathtt{A}')$
$\exists \gamma_{\mathtt{i}} : \gamma_{\mathtt{i}}(\mathtt{R}'(\beta'_{\mathtt{i}})) = \beta_{\mathtt{i}} \quad \wedge \quad \gamma_{\mathtt{i}}(\mathtt{R}'(\Gamma_2)) = \Gamma' \quad \wedge \quad \gamma_{\mathtt{i}}(\mathtt{R}'(\Gamma)) \subset \Gamma_2$

• Lastly, since we're interested in an arbitrary substitution that makes the conditions valid then we have γ as the substitution composing all the γ_i and γ' together, where if $x \in Dom(\gamma_0) \cap \ldots \cap Dom(\gamma_n) \cap Dom(\gamma')$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$:

$\exists \gamma : \gamma(\mathtt{R}'(\beta'_{\mathtt{i}})) = \beta_{\mathtt{i}} \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma_2)) = \Gamma_2 \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma'$
$\wedge$
$\exists \gamma : \gamma(\mathtt{R}'(\Gamma_2, \mathtt{this} : \mathtt{B}')) = \Gamma_2, \mathtt{this} : \mathtt{B} \quad \wedge \quad \gamma(\mathtt{R}'(\Gamma)) \subset \Gamma'_2, \mathtt{this} : \mathtt{B}$

• Therefore $\mathtt{typecheck}(\mathtt{Pr}, \Gamma, \emptyset, \emptyset) = (\Gamma'', \mathtt{R}', \mathtt{A}') \wedge$
$\exists \gamma : \gamma(\mathtt{R}'(\Gamma'')) = \Gamma''' \quad \wedge$
$\gamma(\mathtt{R}'''(\Gamma)) \subset \Gamma'''$

Case $\mathbf{Pr} \Rightarrow \mathbf{E}(\mathbf{E_1}, \ldots, \mathbf{E_n})$:

We want to prove $\Gamma, \mathtt{this} : \mathtt{B} \vdash \mathtt{Pr} : \beta \Rightarrow \mathtt{typecheck}(\mathtt{Pr}, \Gamma, \mathtt{R}, \mathtt{A}) = (\beta', \Gamma \cup \{\mathtt{this} : \mathtt{B}'\}, \mathtt{R}', \mathtt{A}')$
such that $\exists \theta : \theta(R'(\beta')) = \beta$ and $\theta(R'(\Gamma \cup \{this : B'\})) = \Gamma \cup \{this : B\}$
then given
$\Gamma \vdash \mathtt{E} : \beta'' \xRightarrow{\mathtt{by\ \ I.H.}} \mathtt{typecheck}(\mathtt{E}, \Gamma, \mathtt{R}, \mathtt{A}) = (\beta''', \Gamma, \mathtt{R}, \mathtt{A})$
such that $\exists \gamma_0 : \gamma_0(R(\beta''')) = \beta''$ and $\gamma_0(R(\Gamma)) = \Gamma$
and
$\Gamma \vdash \mathtt{E_i} : \beta_{\mathtt{i}} \xRightarrow{\mathtt{by\ \ I.H.}} \mathtt{typecheck}(\mathtt{E_i}, \Gamma, \mathtt{R}, \mathtt{A}) = (\beta'_{\mathtt{i}}, \Gamma, \mathtt{R_i}, \mathtt{A_i})$
such that $\exists \gamma_i : \gamma_i(R_i(\beta'_i)) = \beta_i$ and $\gamma(R_i(\Gamma)) = \Gamma$

With $\beta'' = (\beta_1, \ldots, \beta_n) \to^{B} \beta$, $\beta''' = (\beta'_1, \ldots, \beta'_n) \to^{B'} \beta'$, $i \in \{1, \ldots, n\}$, $R' = \bigcup_i R_i$, $A' = \bigcup_i A_i$.
Since we're interested in an arbitrary substitution that makes the conditions valid then we have θ as the substitution composing all γ_i and γ_0 together, where if $x \in Dom(\gamma_1) \cap \ldots \cap Dom(\gamma_n) \cap Dom(\gamma_0)$ then one type is chosen arbitrarily from one the the defined $\gamma_i(x)$.

□