Smelling of ROSES
ROles - Specification, Exploration and Scrutiny
Master's Dissertation

Azalea Raad
Imperial College London
ar106@doc.ic.ac.uk

Supervised by
Professor Sophia Drossopoulou
scd@doc.ic.ac.uk

Acknowledgements

I would like to thank my supervisor, Sophia Drossopolou, for all her help in refining and inspiring the work in this report. I would also like to thank Susan Eisenbach, not only for her help with this project, but also for the years of support and wisdom I have received from her throughout my degree.

My thanks also go to Krysia Broda for her passion and enthusiasm in the many projects we have worked on together and to my personal tutor Paul Kelly for the foundation he provided for me in my first year. I would also like to mention Margaret Cunningham, Steve Ingram and Amy Allison for providing so much reassurance and help whenever it was needed.

Finally, I would like to thank everyone in the department who has taught, instructed and assisted me throughout my degree, and inspired in me the love I now have for my subject.

Abstract

As computer programs shift towards highly distributed and parallel environments, the importance of reliable and safe communication rises and hence the challenges of safe concurrent computing march to the forefront of modern computing research. One of the most prominent of these is the provision of a verification method for inter-process communication which has proven extremely challenging and has led to one of the most common bugs in concurrent computing - synchronisation bugs.

Session types have been proposed as a means of solving this problem via efficient *type-checking*. Several variants of session types have been studied for various use-cases; these have all attempted to exploit the benefits of type checking by binding the interacting participants to strictly-typed *protocols*, forcing them to conform to the said protocol and hence guaranteeing the communication safety. However, these approaches have various constraints and limitations, and a more suitable solution is sought.

This project specifies *Roles*, a language based on a form of session types suited to dynamic multiparty communication with a number of interesting and useful features. We define the syntax and the operational semantics of *Roles*, present its type system and conjecture about its properties before evaluating it with respect to contemporary approaches.

Contents

Chapter 1

Introduction

Communication has become one of the most crucial aspects of computation since programs are increasingly geared towards distributed data access and parallelism. However, the task of providing a reliable communication system comes with its difficulties. Communication between programs leads to the most common bug in programming; those relating to synchronisation. A programmer assumes the communicating parties will hold a compatible and consistent conversation. Nevertheless, they can easily fail at this task due to incorrect timing. Failure at sending or receiving a message at the correct time by either of the attending parties can lead to a synchronisation bug which can only be detected at runtime.

One of the methods in which participants in a communication system can interact is by sending messages to one another over a *channel*. A channel is a medium of communication between participants over which they exchange messages. If the two parties communicating over a channel do not reach an agreement about the time a message must be communicated, a synchronisation bug can occur. Session types have been proposed to solve the said problem by introducing the benefits of *strong typing* into communication.

A type system is an invaluable component of a programming language since it prevents certain erroneous and undesired program behaviour (type errors). *Session Types* as proposed by Kohei Honda in [30] yield a mechanism of safe communication by producing a typed foundation for structured and well-behaving communication-based programming. Session types provide us with a means of ensuring the safety of communication by specifying strictly-typed protocols that govern the behaviour of a communication system and guarantee the expected behaviour by forcing the conformance to the specified protocols.

Session types have been studied over the last decade [30, 55, 5, 16, 24, 33, 9] for a wide range of process calculi and programming languages, focusing on binary (two-party) sessions. However, in order to specify the behaviour of larger and more complicated systems, multiple channels must be defined usually with the same session type since they specify the behaviour of a single channel (involving two parties) at a time. They force us to define dedicated channels between each pair of communicating parties and hence result in complex systems. Thus, understanding the communication, reasoning about the system and establishing the safety properties becomes much more challenging.

Global session types [32] provide a solution to this problem by specifying the communication protocol in terms of participants, unlike dyadic session types where the interaction protocol was defined per channel. With this approach reasoning about and understanding the communication within large systems is much easier, since they reduce the number of session types declared considerably. Nevertheless, one of the limitations of global session types is that they do not cater for dynamic joining/leaving of the participants. Once the communication starts, the number of parties must remain the same throughout the communication. No participant is allowed to leave the system until the communication session ends and newcomers will be prevented from joining an ongoing session. Consequently, the number of participants in a communication system must be known upon initiation of the communication and this approach does not allow for the number of parties to be unknown. We believe that these are invaluable properties of any communication system given their prevalence in many use-cases and it would be highly favoured for any communication language to support them.

Neither dyadic nor global session types provides us with flexible enough abstractions in order to model large systems with sophisticated patterns of interaction. They are also limited in their ability to express broadcast semantics. Communication between parties with multiple members (many-to-many, or many-to-one) is possible in both approaches, but only when the exact number of participants is known. Global session types requires this information statically, but even in the looser case of session types it is extremely hard to receive on large numbers of channels simultaneously, and equally difficult to create channels during a conversation. Very few attempts ([46]) at implementing session types have tried to capture this behaviour.

1.1 Aims

Although dyadic and global session types have taken the right step towards safe communication by introducing the benefits of typing into communication, they both have their constraints and limitations. This project introduces the notion of roles [27] for modelling multi-party communication systems. *Roles* allow for an arbitrary number of participants as well as dynamic leaving of members so long as they conform to the type-safety constraints. Communication between different parties is defined in a *conversation* within which different roles are introduced. Conversations also specify channels between the roles and their types. Communication is broadcast by definition and caters for many-to-many communication since channels are defined between different roles and a message addressed at a particular role is aimed at every participant of that role. We believe that the notion of *Roles* will provide us with a highly flexible abstraction language when used to model large complicated multi-party systems, and that it will result in far more concise definitions compared to those of existing approaches.

The aim of this project is to introduce a new language for modelling multi-party communication systems based on the ideas put forward in [27], referred to as *Roles*. *Roles* allow for an arbitrary number of participants as well as dynamic joining or leaving of members so long as they conform to the type-safety constraints. Communication between different parties is defined in a *conversation* within which different roles are introduced. Conversation also specifies channels between the roles and their types. Communication is broadcast by definition and caters for many-to-many communication since channels are defined between different roles and a message addressed at a particular role is aimed at every participant of that role. Broadcast semantics is an important feature of a language since it helps us avoid code duplication and repetition in channel specification when the same message is to be sent to or received from a group of participants (cf. sections 2.5.4 and 2.6.4). We believe that the notion of *Roles* will provide us with a highly flexible abstraction language when used to model large, complicated multi-party systems, and that it will result in far more concise definitions compared to those of existing approaches.

The rest of this report is organised as follows. In chapter 2 we will provide some background information on the communication safety issues and will discuss existing solutions. We will analyse the work by Honda et al on various incarnations of session types as well as other works in a similar vein. Using a standard set of examples which test the expressiveness of a formalism for interprocess communication, our analysis will weigh the

strengths and weaknesses of these approaches.

In chapter 3 we familiarise the reader with our notion of roles in broadcast communication. We then proceed to define the syntax of the *Roles* language, specify its operational semantics and lastly present its type system together with its conjectured properties. Finally, in chapter 4 we evaluate each component of the *Roles* language specification in turn, discuss the challenges of the project and ultimately suggest further areas of work to develop the *Roles* language.

Chapter 2

Background

This chapter will introduce the motivating work for this project. We begin by assessing the ways in which concurrent computation can be modelled and analysed, before introducing some of the problems which the first session types work tried to tackle. We then introduce both *dyadic session types* and *global session types*, showing their limitations. Finally, we consider other, contemporary approaches to the problem of safe concurrent communication.

2.1 Models of Concurrent Computation

Concurrent computation is now the backbone of many fields of modern computer science, including multicore programming [48, 44], cloud computing [23], compiler optimisation [14], quantum computing [42] and is considered a key component of computing in education [11] as a result.

Safe communication between programs is growing in importance as programs are progressively shifting towards highly distributed and parallel environments. Nevertheless, providing safe and reliable communication has proved a challenging task and has attracted the focus of many researchers. The push towards concurrent computation is encouraged by two equally landmark changes in the last two decades - the move from serial programming to parallel programming that marked the end of Herb Sutter's 'free lunch' [47]; and the massive expansion of the Internet which now forces programs to routinely engage in massively distributed communication across a vast and unreliable network.

These two factors - the Internet's growth and the increased focus on parallel computation on local machines - has forced programmers to work in more complicated domains than they are used to, where operations that

were previously taken for granted - such as information exchange between two processes - is now fraught with complication. The desire to simplify, abstract or otherwise remove these complications for the programmer is what drives much research into concurrent computation today.

2.1.1 Concurrent Paradigms

Methods of communication between concurrent components in a program fall into two main categories.

- **Shared Memory** In this approach a common location in memory is available to processes, perhaps analogous to a shared message box of sorts. Shared memory can be seen as a powerful approach to concurrent communication, since the shared regions can be used as regular memory to the processes using them. However, because of the need for a universally accessible storage area these systems cannot be distributed outside of a single physical machine, since there is no suitable location for the shared memory area. This makes shared memory a far less desirable approach in the long-term, as computing tends towards a more distributed nature.

- **Message Passing** This paradigm provides a system of messaging that processes can use to send information to one another. These messages may contain information on critical sections, locking states or otherwise - the point is that these messages are the only means by which processes can exchange information. Much work has been done in the field of message-passing, particularly work into automating the conversion of serial programs to concurrent systems employing messaging [54, 4]. This approach is employed in a few programming languages and the vast majority of calculi. Our notion of *Roles* is a part of this paradigm, although when used with the object-oriented paradigm, inter-thread communication can take place via shared memory.

Although exploratory work has been done to unite the two approaches [35], message-passing is often found to be easier to reason with due to its familiar analogy of message exchange in real world situations. Some research has been done to compare the two that backs up this theory [36].

2.1.2 Analysis of Message Passing Techniques

In order to reason about message-passing ideals, various process algebrae have been employed. Process Algebrae are useful formalisms for exploring properties of communications systems such as behavioural equivalence and synchronisation. In general, an algebra is a mathematical formalism for performing operations on sets of values. In the case of process algebra, the values are the processes themselves. The term *algebra* implies certain properties of the operations that may be performed on these values - properties such as commutativity and distributivity.

Well-known process algebrae such as CSP[28] and CCS[37] have been used for years, but more recent calculi, particularly the π-calculus, are proving extremely useful in the analysis of concurrent systems. The π-calculus [40, 41] was proposed as the next step after CCS, developed as it was by Robin Milner who was the original proponent of CCS. Its proposal came in two papers [40, 41], and then formed the basis of a book by Milner on the topic [39], and has since been extended to such diverse fields as cryptography [3], molecular biology [45] and even business theory with the short-lived *Business Process Modelling Language*.

One key step forward that the π-calculus took from its predecessors was the concept of *process mobility*. The π-calculus allows for channels themselves to be communicated as values between processes, a property called *session delegation*. That is to say, a process can pass a conversation-in-progress to another process who continues the conversation in their place. Delegation allows a programmer to dynamically distribute a single session among multiple processes in a well-structured way.

The π-calculus has been shown to be a *universal model of computation* [38], a property also known as *Turing completeness* that is shared with the lambda calculus and Turing machines. This simply means that the π-calculus can be used to produce the result of any arbitrary calculation. Its minimality and completeness has made the π-calculus a popular choice for describing communication systems and has been widely used as a base to describe the syntax of several session types variants.

Basic Pi-Calculus Syntax

The basic constructs of the π-calculus syntax are outlined below where P and Q are processes.

- $P \mid Q$ defines concurrent action. Both processes P and Q may take action.

- $c(x).P$ is an input. Process P receives an input on channel c and binds it to the variable x.
- $\bar{c}\langle y\rangle.P$ is an output. Process P outputs message y on the channel c.
- $!P$ is *replication*. A replicated process can create a copy of itself - that is, another P.
- $(\nu x)P$ is the allocation of a new constant x within process P. Constants are new channels of communication in π-calculus .
- 0 is the null process. A process that is equal to this has finished executing.

2.2 Communication Safety Issues

When two parties attempt to exchange information with one another, two common problems often occur. The first of these occurs when both participants are expecting to receive a message from the other, resulting in a lack of progress by either. The second of these occurs when there is a conflict between the format or data type of a piece of exchanged information.

We now illustrate these two problems using a subtle variation on a common situation; we describe a collaboration pattern that appears in many web service business protocols. Figure 2.1 shows the sequence diagram, for the protocol which models the purchase of a book, between the seller of the book and a potential buyer.

First the buyer sends the title of the book which he is interested in to the seller. In some cases, the seller might have different versions of the same book in stock with different prices - he might have the same book pressed by different publishers or he might have it in both paperback and hardcover. So in order for the seller to provide the buyer with the best quote, the buyer must provide the seller with the highest amount he is willing to pay. The seller then will filter the options accordingly, and communicate back all available options priced less than or equal to the highest price indicated by the buyer. If the buyer is happy with the recommended price, then he sends his address to the buyer together with his final choice and the seller informs him of the delivery date. If the buyer is not interested in any of the suggested options, the communication ends.

Now, assume that the connection between seller s_1 and buyer b_1 is established and that the following scenario takes place.

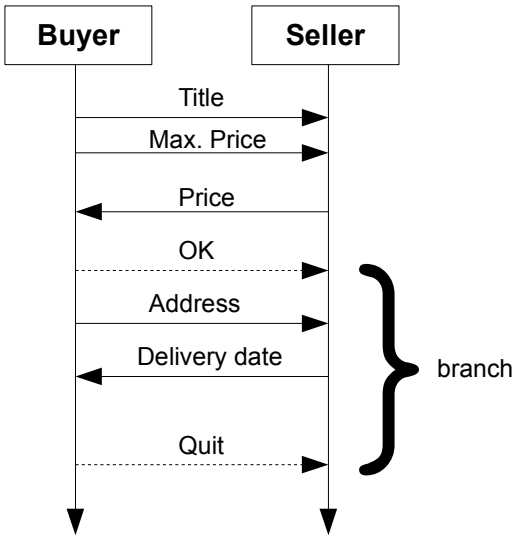

Figure 2.1: Sequence diagram for purchasing a book

- b_1 is interested in *Atlas Shrugged* by *Ayn Rand* and hence he sends the "Atlas Shrugged" message to s_1.
- b_1 is not aware of s_1's sale policy and he does not provide him with maximum price he's willing to pay towards postage and packaging. Hence he is waiting for the price of the requested item.
- s_1 receives this message. However, he is still waiting for a secondary message from b_1 containing the highest amount he is willing to pay.
- Both s_1 and b_1 are waiting for each other without progress, hence a deadlock has occurred.

The above scenario failed due to the absence of a common protocol or an agreement between the two parties. If buyer b_1 was aware of s_1's sale policy; that is to say, if at some point in the past prior to the program execution at either site an agreement had been reached by the two of them with regards of the flow of the transaction, this problem could be easily avoided.

Now consider a modification to the above example. Suppose now that s_2 does not require the buyer to communicate the highest desired price. Now consider b_2, who wishes to purchase a book but will not pay more than ten

pounds. b_2 is expecting an integer value to compare with his maximum price. The following scenario can violate the safety of the communication.

- b_2 is interested in *Atlas Shrugged* by *Ayn Rand* and hence he sends the "Atlas Shrugged" message to s_2.
- s_2 receives this message. He currently sells this book at RRP of seven pounds. He sends a message back to b_2 containing the *string* value "seven GBP".
- b_2 receives s_2's message and he compares it to the *integer* value 10.
- The program on b_2's side halts prematurely since the two values are incompatible.

Again, the failure of this scenario is down to the lack of a consistent contract between b_2 and s_2. If a protocol had been designed to govern the behaviour of both the buyer and the seller by which both parties were bound to abide by this problem could have been circumvented. A simple solution to the second example would be a protocol which restricted the types of the data exchanged.

Session types provide a solution to these problems and will be discussed in the next chapters.

2.3 Session Basics

A *session* is a sequence of interactions between two communicating parties also referred to as a conversation [32]. For two parties to communicate, first a connection must be established between them. At this point the session starts and the communicating parties may interact with each other using a channel. As the name suggests, a channel is a communication medium over which participants may interact by sending and receiving messages. Each session is composed of a series of communications between the attending parties interleaved with local computations on each side. Communication between the parties is in the form of sending and receiving values.

Throughout the interactions between two parties, the underlying type system (Session Type) enforces a perfect correlation between sending and receiving actions. In other words whenever a party sends a value, the other receives it and vice versa. This property of the session is referred to as the *duality* of the communication.

2.4 Session Types Design Space

When designing a language for describing interprocess communication, there are many decisions to be made about what properties and functionality that language will have. This section explores some of the decisions that must be made, and that we will have to consider in our design of *Roles*.

1. **Underlying Paradigm** Broadly speaking, there are three underlying paradigms which one can adopt when designing a language for session types. The first is the functional paradigm, as demonstrated by Sackman et al in [46] and Vasconcelos et al in [52]. The second approach is by way of the π-calculus , an approach favoured by Honda et al in their work in [32]. Finally, some approaches attempt to employ an object-oriented paradigm. This work can be seen in work by Dimitris et al in [17]. The choice of paradigm is important because it affects future use of the work, and directly impacts where and how it may be implemented.

2. **Progress** The *progress* property holds true for a communication system if that system can never reach a state of *deadlock*; that is, all communications eventually succeed. Progress is a desirable property for communication systems as it makes them considerably more reliable - however, ensuring this property places a heavy burden on the type system. The issue to consider here is whether or not to ensure progress in spite of this.

3. **Synchronicity** Communication between two processes is either *synchronous* or *asynchronous*. Synchronous communication (also known as *blocking* communication) forces each participant to wait upon sending a message until its recipient confirms that the message has been received. Asynchronous communication (non-blocking) does not require its participants to wait; instead the sender may continue execution. Synchronous protocols can cause processes to wait for unnecessarily long periods of time and become complicated if a process fails, which asynchronous protocols do not suffer from.

4. **Delegation versus Higher Order Sessions** The question to consider here is whether or not the language will support higher order primitives such as *delegation*. Languages who allow for delegation are those who let the communication channels to be exchanged as values between processes. Alternatively, some languages allow for higher order sessions, in which sessions can be passed as parameters or stored as

fields or variables, rather than simply passing the channel to another process. An example of the former is the work Honda et al in [32], and the latter can be seen in [21].

5. **Branching** There are two approaches to branching - the first is to explicitly label the branches in the specification, where a branch is of the form *label* : `P`. In this approach, the selecting process will decide which course of action to take based on the label. The second is to use a kind of pattern matching on the type of values passed. For instance, a branching process may be outputting an `integer` on one branch and a `boolean` on the other. The selecting process then decides which branch to take based on the type of value it is expecting to receive. Again, an example of the former approach is demonstrated in the work of Honda et al [32]. The latter approach is demonstrated by [17].

6. **Channel Specification** When defining a channel we must specify what we are defining it in relation to. Some models of concurrent computation define a channel with respect to the participants, where there is a dedicated channel for each party and any other participant who wishes to communicate with the said party must pass messages through this channel. An alternative approach will be to define a channel for each pair of communicating parties; each participant now sends and receives on multiple channels to and from multiple participants.

In the subsequent sections, we will examine several approaches to session typing and other information exchange protocols and and as we do so we will consider the choices they made about the above factors.

2.5 Dyadic Session Types

Session Types as proposed in [30, 55, 29] provide us with means of planning and defining conversations between concurrent processes and enabling them to cooperate and communicate in a well-structured way.

Session types have been introduced into different settings from variants of π-calculus [29, 30, 55, 49, 6, 5, 26] to CORBA [50], ambients [24], object-oriented programming languages [19, 15, 13, 17, 50], functional languages [25, 51, 52] and W3C standard description language for Web Services,CDL [2, 9, 10, 43, 31]. Applications of session types range from operating systems [22] to web services [43].

2.5.1 Proposed Solution

Session types solve the safety issues discussed in section 2.2 by defining protocols to govern the communication channels and forcing the parties sharing the communication channels to conform to the specified protocol. They treat conversations as types and type-checking then enforces the safety of the communication by asserting that both parties speak the same protocol. In other words, Session Types are a means to establish conformance to protocols in distributed applications and guarantee the following:

- **Safety** - Interactions between the attending parties in the session never results in a communication error.
- **Progress** - Channels are deadlock free and are used linearly.
- **Session fidelity and predictability** - communication sequence conforms to the one specified in the session type.

2.5.2 Design Space

With regards to the factors discussed in section 2.4, dyadic session types have made the following decisions.

1. **Underlying Paradigm** Dyadic session types adopt the π-calculus paradigm.
2. **Progress** As stated above, dyadic session types ensure the progress property.
3. **Synchronicity** The language of dyadic session types as proposed in [30] opts for synchronous data sending/receiving.
4. **Delegation versus Higher Order Sessions** Dyadic session types support channel delegation, that is, they allow for the name of the channels to be communicated as values.
5. **Branching** Dyadic session types uses a label-based branching system, as seen in the applied π-calculus .
6. **Channel Specification** Channels are specified for each pair of communicating parties. If a pair of participants in the system wish to communicate, a dedicated channel will be defined for their interaction.

2.5.3 Buyer Protocol Example

In order to illustrate the functionality of the session types, a slight modification of the example in section 2.2 is introduced. Figure 2.2 shows the sequence diagram for the protocol which models the purchase of a book between the buyer and seller. First the buyer sends the title of the book which he is interested in, then the seller communicates back the price of the book. If the buyer is happy with the recommended price, he sends his address to the buyer and the seller informs him of the delivery date. If the buyer is not interested in the price, the communication ends. The session type of this protocol is described below from buyer's perspective.

$$\texttt{buy-book = !Title.?Price.!}\{ok \triangleright \texttt{!Address.?DeliveryDate},\ quit \triangleright \varepsilon\}$$

The *dual* of the above type from the seller's perspective is given as follows.

$$\overline{\texttt{buy-book}} \texttt{ = ?Title.!Price.?}\{ok \triangleright \texttt{?Address.!DeliveryDate},\ quit \triangleright \varepsilon\}$$

ok and *quit* are used as *labels* to mark alternative courses of action depending on certain conditions and $\triangleright$ is used to denote branching. In other words, if the buyer is interested in the book at the recommended price, he will *select* the *ok* branch and send his address to the seller; otherwise, he will *select* the *quit* branch and the conversation ends. Since the behaviour of the buyer is not known statically while specifying the session type, all possible actions are listed using branching and labelling.

2.5.4 Auction Example with Dyadic Session Types

The previous example consisted of two participants only. The next step would be to consider more detailed examples with more participants. One suitable scenario would be an ebay-style auction as described in [27].

In this example, there are three different kinds of participants - auctioneer, bidders and the audience. There is a single auctioneer at any given time and for simplicity we assume for now that there are m bidders and n members of the audience when the auction starts and this number remains fixed throughout the auction. The auctioneer announces the item to the audience and the bidders and at this point the bidders can bid for the item. None of the bidders can see who else is bidding, nor hear each other's bid.

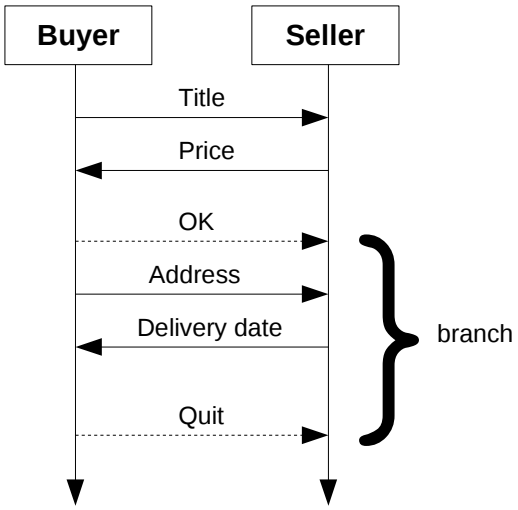

Figure 2.2: Sequence diagram for purchasing a book

Once the auctioneer has received a bid from each of the bidders, he compares them against each other and determines the highest bid. He then announces the winning bid to the audience as well as the bidder. The auctioneer then announces the next item on bid and the same chain of events take place.

The channels of this protocol can be specified using dyadic session types as follows.

`Auctioneer-Audience-1` = μT.`!Item.!Winner`T
...
`Auctioneer-Audience-n` = μT.`!Item.!Winner`T

`Auctioneer-Bidder-1` = μT.`!Item.?Bid.!Winner`T
...
`Auctioneer-Bidder-m` = μT.`!Item.?Bid.!Winner`T

As the name suggests, the `Auction-Bidder-k` channel is between the auctioneer and the `k`th bidder from the auctioneer's perspective. The auctioneer announces the item and then listens for his bid. Once he has received a bid from all bidders, he announces the winning bid. The `Auctioneer-Audience-k`

channel is between the auctioneer and the `k`th member of audience from the auctioneer's point of view and can be justified similarly. Members of audience do not take part in the bidding process and only listen for messages regarding the items on auction and the winning bid.

2.5.5 Discussion

One of the drawbacks of this approach is the large number of channels to be defined. We need n distinct channels between the auctioneer and each member of the audience and m distinct channels must be defined between the auctioneer and each bidder. In this example members of the audience do not need to communicate with the bidders. However, in another scenario where this was necessary, an additional $m \times n$ channels must be present to ensure each member of audience can talk to each of the bidders. This leads to an extremely large number of channels and thus reasoning about this system is complicated.

Additionally, although the `Auctioneer-Audience-k` channels have identical behaviour, each channel must be explicitly specified. Thus, we end up with duplicated code that cannot be avoided.

Furthermore, the number of participants is fixed throughout the communication and dynamic joining/leaving of the members is not allowed. Once the session starts, existing members cannot leave the system until the end of the conversation and newcomers are stopped from joining the ongoing conversation.

Global session types have been proposed as a solution to some of the limitations of dyadic session types and will be discussed in the next section.

2.6 Global Session Types

Global Session Types are proposed by Honda et al in [8, 9] and explored more fully in [32]. Rather than dyadic session types abstracting only a peer-to-peer communication, global session types abstract an interaction of multiple participants into a single global scenario. The syntax remains similar in form to dyadic session types but has an extra expressive power in its ability to capture multi-party interactions, where a single session type is shared among all who participate in the interaction. This shared type forms a common protocol; an agreement between the participants which can be verified via type checking, as with its dyadic form.

2.6.1 Proposed Solution

Global session types solve the problems discussed in 2.5.5 by specifying the communication protocol in terms of participants rather than the channels. In this approach there are distinct channels dedicated to each of the participants and communication with each participant take place over his dedicated channel. For instance, assume that channel c_1 has been dedicated to participant p_1. When any of the participants want to send a message to p_1, they do so by communicating the message over channel c_1. Global session types reduce the number of channels required considerably and hence result in cleaner and more manageable systems which in turn makes them easier to understand and reason about. However, in some situations a participant may need to possess more than a single channel. When a participant is expecting messages of the same type from two distinct participants, it is not guaranteed in which order these messages will arrive and hence they can be easily confused since they have the same type. The two buyer protocol as described in [32] is a good example of this situation and is described in section 2.6.3.

2.6.2 Design Space

With regards to the factors discussed in section 2.4, global session types have made the following decisions.

1. **Underlying Paradigm** Global session types use the π-calculus paradigm, similar to their dyadic form.

2. **Progress** Global session types have been shown to ensure the progress property. A proof is stated in [32].

3. **Synchronicity** Global session types language as proposed in [32] implement asynchronous message passing. When a message is sent over a channel, it is added to a message queue. Thus the sender is not blocked and asynchronous message passing is achieved.

4. **Delegation versus Higher Order Sessions** The language of global session types allows for session delegation.

5. **Branching** Like dyadic session types, global session types also support label-based branching.

6. **Channel Specification** Channels are specified in terms of participants. In other words, each participant in the communication system

is allocated a channel and other parties who wish to contact him can do so by sending messages over this channel.

2.6.3 Two Buyer Example

This example is similar to the one described in 2.5.3 with two buyers interested in the book. Figure 2.3 shows the sequence diagram of this protocol. The first buyer (`Buyer1`) sends the title of the desired book to the `Seller`, `Seller` in turn informs both buyers of the price. At this point, `Buyer1` tells `Buyer2` how much he is willing to contribute towards the full price. If he agrees to pay the rest, he sends the delivery address to the buyer and the buyer confirms the delivery date. Otherwise, the communication terminates. Honda et al specify this protocol thus.

```
Buyer1 = ā[2,3](b1, b2, b2', s).s!<''War and Peace''>;
                                b1?(price);
                                b2'!(price div 2);


Buyer2 = a[2](b1, b2, b2', s).b2?(price);
                                b2'?(contribution);
                                if(price - contribution ≤ 99) then
                                   s◁ ok;
                                   s!<address>;
                                   b2?(x);
                                else s◁ quit; 0


Seller = a[3](b1, b2, b2', s).s?(title);
                                b1, b2!<price>;
                                s ▷ { ok:   s?(x); b2!<date>;
                                      quit:  0 }
```

In a system with n participants, the $\bar{a}$`[2..n](s̃)` notation begins a new session by sending a list of newly generated session channels, `s̃`, to the other *n-1* participants. Each participants is of the form `a[k](s̃).`Q_k where k takes some value in the range `[2..n]`. All participants receive the list `s̃`, and the communication now takes place among the parties listed.

In this example, `Buyer2` receives two values one after the other one from the `Seller` - that is the price of the desired book - and one from `Buyer1` which is the amount he is willing to contribute. `Buyer2` can easily confuse the two values since they are of the same type and there is no guarantee that they will arrive in a specific order. If we were to use a single channel for sending messages to `Buyer2`, depending on the order the messages arrive we might have erroneous results and lose the linear usage of a channel. Hence, we use two separate channels for passing messages to `Buyer2`, one to receive messages from the `Seller` (b_2) and the other to receive messages from `Buyer1` (b_2').

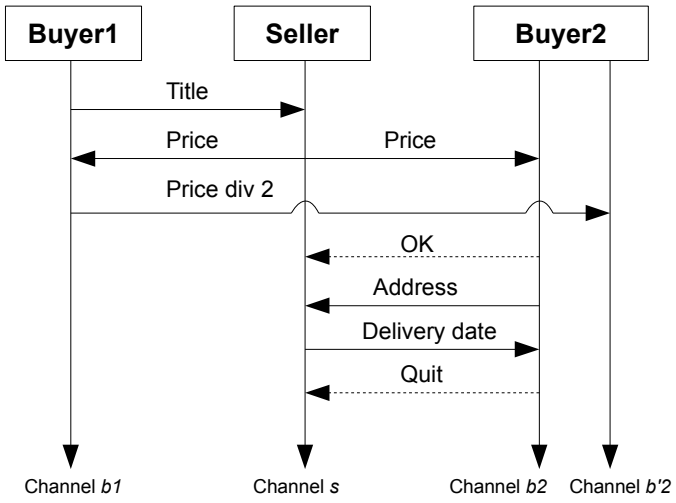

Figure 2.3: Sequence diagram for purchasing a book with two buyers

2.6.4 Auction Example with Global Session Types

The protocol described in the auction example in 2.5.4 can be specified using global session types calculus given in [32].

The protocol for the auctioneer is specified as follows.

```
Auctioneer = ā [2,3,...,m+n+1](s, b1, b2, ..., bm, a1, a2, ..., an).
               a1, a2, ..., an!<item>;
               b1, b2, ..., bm!<item>;
               b1, b2, ..., bm?<bid>;
               a1, a2, ..., an!<winner>;
               b1, b2, ..., bm!<winner>;
               Auctioneer;
```

Similarly, the audience protocol can be written thus.

```
Audience_1 = a[2](s, b1, b2, ..., bm, a1, a2, ..., an).
               a1?(x);
               a1?(y);
               Audience_1;


Audience_2 = a[3](s, b1, b2, ..., bm, a1, a2, ..., an).
               a2?(x);
               a2?(y);
               Audience_2;

               ...

Audience_n = a[n+1](s, b1, b2, ..., bm, a1, a2, ..., an).
               an?(x);
               an?(y);
               Audience_n;
```

Finally, we can specify the protocol governing the bidders' behaviour as outlined below.

```
Bidder_1 = a[n+1+1](s, b1, b2, ..., bm, a1, a2, ..., an).
                b1?(x);
                s !<myBid>;
                b1?(y);
                Bidder_1;


Bidder_2 = a[n+1+2](s, b1, b2, ..., bm, a1, a2, ..., an).
                b2?(x);
                s !<myBid>;
                b2?(y);
                Bidder_2;

                ...

Bidder_m = a[n+1+m](s, b1, b2, ..., bm, a1, a2, ..., an).
                bn1?(x);
                s !<myBid>;
                bn?(y);
                Bidder_m;
```

2.6.5 Discussion

Global session types provide a notion for describing scenarios involving multiple peers. In this approach interactions involving multiple parties are abstracted as global types that plays the role of a shared agreement among communication parties. However, this approach has its own drawbacks.

Global session types do not cater for dynamic joining or leaving of the participants. Once the session begins, the number of parties remains the same all through the session and no participant can leave the system. Newcomers cannot join the system even if they have identical behaviour to those already defined.

For instance, in the auction example, any attempt by new bidders to join the system would fail although their behaviour is the same as the existing bidders. This is because each participant in the system described by global session types is allocated a dedicated channel to receive messages from other parties and dynamic joining requires dynamic channel generation for newcomers which is not supported by this notion.

Consequently, the number of participants has to be fixed prior to the initiation of the session and one cannot create a session with an unknown number of participants. These are all invaluable properties which cater for a large number of use-cases.

Although global session types when used in multiparty scenarios provide a much more concise description of the protocol in comparison to dyadic session types, they are not completely free of repetition. In the auction example, the behaviour of each of the audience members and bidders had to be explicitly specified despite their identical behaviour.

The question that comes to mind is whether or not this can be avoided at a higher level? That is, if duplication occurs only when describing the protocol in π-calculus and if the user is provided with high level constructs, would that let her define such processes using a mapping function and avoid duplication?

Global session types make the behaviour of the participants transparent at a higher level than they should; the detailed behaviour of the process is specified while defining the process rather than pushing the behaviour of the participant into the relevant part of the higher level program (which is within programmer's responsibilities to determine) and defining the process in terms of a sequence of events (input/output actions) that may take place.

The limitations of global session types have not been explored fully in any other languages. However, other approaches have been proposed in order to formalise multi-peer communication systems and will be discussed in the following sections.

2.7 Conversation Types

In [53, 7], Caires et al propose a process calculus based on a notion of *conversation* and *conversation context*. The work, primarily aimed at the field of service-oriented computing, has some parallels with the work on session types.

Conversation Contexts

A conversation context is a distributed medium for communication between processes. The context is distributed between multiple processes, and those processes can communicate and interact with any other process in the same context. Processes are aware of these contexts - Caires introduces a construct whereby a process can obtain information on the context it is live in. For a process P, the construct $here(x).P$ represents this awareness - x becomes bound to the name, n, of the context that P is currently in. P continues execution, with x bound now to n.

Communication via Conversation

Caires outlines three ways in which a processes can interact with one another.

- Context-based communication, where processes communicate internally within the context.
- Endpoint-to-endpoint communication, where two endpoints (client and server, perhaps) communicate across a context.
- Inter-contextual communication, where a context interacts with a larger, enclosing context.

This last point also highlights that contexts can be *nested* within each other. A nested context appears as a normal process to the enclosing context, which Caires says allows for more complex subsidiary services and composite processes. This is of most relevance to the service-oriented aspects of the research.

2.8 Orchestration

In computing, the term *Orchestration* refers to the co-ordination of web-services in terms of the functionality they offer and the information they are able to exchange. This way of thinking about web services has been used most prominently in the *service-oriented architecture*, where services which express fine-grained functionality can be combined together to create a more complex composite web service. Many languages exist for specifying web services this way, and many of them share traits or motivation with the work on session types. We consider a few notable examples in this section.

2.8.1 Business Process Execution Language

The Business Process Execution Language, or BPEL, is a language that specifies how web services interact via the internet. The BPEL is one of a family of languages concerned with the so-called task of *programming in the large*; the coarse-grained tasks of interaction such as sending and receiving messages and coping with failure of communication between participants. It grew out of a surge in business modelling languages, particularly *WSFL* and *XLANG*, developed by Microsoft and IBM respectively and from which BPEL is derived.

BPEL defines relationships between processes, rather than specifying anything concrete about execution or declaring any tasks to be performed. Figure 2.4 shows an example of this, specifying a *business partner relationship* between two processes. This example is derived from *WS-BPEL - A Primer* [1]. According to the primer, the concept of partner links represents 'typed connectors' that 'specify... port types the process offers'. Note the similarity to session types here - the idea that communication is fixed by the data types it permits, and that these relationships should form the backbone of the system specification.

```
<partnerLinks>
   <partnerLink name="ClientStartUpLink"
      partnerLinkType="wsdl:ClientStartUpPLT" myRole="Client" />
</partnerLinks>
```

Figure 2.4: A simple example of a relationship between two processes in WS-BPEL.

2.8.2 The Orc Programming Language

The Orc programming language, proposed by Cook and Misra in their position paper [12] and later explored more fully in [34], is described as being aimed at 'distributed and concurrent programming'. It is both a general purpose programming language for writing and executing applications, and a specification language for concurrent interactions, the aim being to add high-level programming concepts to a strong foundation that is not unlike the π-calculus in its structure.

Simple Orc Example

ORC-EXAMPLE

```
1  Prompt("Enter a name.")
2  | Prompt("Enter another name.")
3  >name>
4  "Hello, " + name
```

Figure 2.5: A simple example of process composition in Orc

A program in Orc has several features that are similar in both form and function to the π-calculus . Fig. 2.5 shows a sample program written in Orc. Lines 1 and 2 demonstrate the composition operator |, similar to parallel execution operators in process algebrae. The operator causes both `Prompt` expressions to be executed, presenting two input boxes to the user.

The >> infix operator waits for its left-hand side to publish a value before executing the right-hand expression. In this case, the operator has been embellished by placing a variable name, *name*, in between the two > symbols. This causes the published value to be caught and stored in the variable, and its subsequent use is clear in the lines below.

Conclusions

In this chapter we have provided the background theory and motivation for our work on *Roles*. We looked at concurrent computation and its inherent problems, along with early attempts to solve them. We also looked at modern-day work involving session types. In particular, we analysed dyadic session types and global session types as ways of ensuring safe communication between parties. We evaluated their strengths and weaknesses and looked at related work in neighbouring fields, including web development and business theory. This work motivates the development of *Roles* by demonstrating the varied ways solutions have been attempted, as well as revealing their shortcomings. In the next chapter we will develop this understanding by introducing our proposed language of *Roles*.

Chapter 3

Specification

This project is inspired by the current work on multi-peer session types [32, 53, 7] and is a continuation of the work by Giachino et al in [27].

While dyadic session types specify the communication protocol of a system in terms of the channels between each pair of peers and global session types describe the communication by outlining each individual's (participant's) protocol, Giachino et al propose a language in which the behaviour protocol is defined for each *role*. We have named this language *Roles* after one of its integral concepts. In the subsequent sections, we will familiarise the reader with the notion of conversations, present *Roles* syntax, its operational semantics and finally its type system.

3.1 Introduction to Conversations and Roles

In the context of the *Roles* language, a *role* is a collective name given to a group of participants who share identical behaviour. However, multiplicity is not a defining characteristic of a role and some roles might have a single participant. In other words, the cardinality of a role varies between one to many. For instance, in the auction example discussed earlier in section 2.5.4, there are three distinct roles, namely the *auctioneer* role, the *bidder* role and the *audience* role. The auctioneer role consists of a *single* participant -"The Auctioneer" - at all times, while each of the bidder and audience roles might have one to many participants at any given time.

A communication system is defined in terms of a *conversation* which consists of several roles and channels. Each channel is dedicated to communication between a pair of roles which is clearly specified in the definition of a channel. Also attached to each channel is its *session type* which is the

protocol governing the behaviour of one side of the communication only. The behaviour of the other end-point is given by the *dual* of the said session type which can be obtained by replacing each ! with a ? and vice versa. The choice of the end-point whose behaviour is described by the channel session type is arbitrary and does not affect the conversation.

The end-point session type is defined for the role collectively rather than expressing the behaviour of each individual separately. That is to say, all participants in a role share the same behaviour and they are all obliged to abide by the protocol specified by the role. The specified protocol is the "terms and conditions" of using the channel and a role can continue communicating over a channel so long as *all* of its participants agree to conform to the specified session type.

If the session type specified by the channel indicates that the next action for one of its constituent roles is to send a value then *all* participants of that role must perform a send action over that channel. Similarly, when the receiving action is performed at the other end-point, the same number of values must be received. This requirement has been captured in the *Roles* language in [27] and will be discussed later.

3.2 Syntax

The syntax of the *Roles* language is as depicted in Figure 3.1. The syntax has been divided to *runtime syntax* - indicated by the grey boxes - and *source language syntax*. Runtime syntax refers to expressions that are only produced as a result of the reduction process and do not occur in the source code. This includess the heap, threads (processes) and addresses in the heap. Source language syntax on the other hand refers to expressions found at source level. Below we list various elements of the *Roles* syntax and give a brief explanation of their usage.

3.2.1 Types

The metavariable `t` refers to the possible types of expressions and ranges over class names (`ClassId`) and conversation names (`ConvId`).

Session types (`S`) are the types of communication channels where a sequence of session parts (! and ? symbols for *output* and *input* actions respectively) describe the communication protocol.

Each session type `s` has a dual type $\overline{\texttt{s}}$, obtained by replacing each ! action by ? and vice versa.

(type)	t::= ClassId | ConId
(Program)	prog ::= class × con
(class)	class ::= ClassId → ClassId × (FldId → t) × (MethId → meth)
(method)	meth ::= t m ($\overline{\text{t x}}$) requires (CVId × ChId → S){e}
(conversation)	con ::= ConId → RoleId × chan
(channel)	chan ::= ChId → (RoleId × RoleId × S)
(session)	S::= ?(C).S | !(C).S | ϵ
(expression)	e::= x | v | e; e | this | new ClassId() | new ConId() | e.f | e.f := e | e.m ($\overline{\text{e}}$) | e.ChId.send(e) | x = CVId.ChId.receive() in e | e.r.join() | e.r.leave() | e.start()
(value)	value::= null | ObjAddr | ConvAddr | EOM
(thread)	P::= (e, ProcAddr) | P|P
(heap)	heap::= ObjAddr ↦ object | ConvAddr ↦ conv |(ProcAddr × ConvAddr × ChId) → (int×int×int)
(object)	object ::= ClassId → (FldId × value)
(conv. instance)	conv ::= ConId × (RoleId → ProcAddr*) × (ChId × int → value*)
(Channel ID)	ChId ::= ch | d(ch)

Figure 3.1: Syntax, where syntax occuring at runtime only is shaded

3.2.2 Class Declaration

Programs are defined as a set of classes and conversations. Each class representation consists of a `ClassId` denoting the name of the class; a `ClassId` indicating the name of the super class; a sequence of fields of the form `FldId` $\times$ `t` where `FldId` stands for the name of the field and `t` refers to its type; and a sequence of methods.

3.2.3 Method Declaration

Each method is represented in the standard object-oriented style consisting of a return type, a method name, a list of arguments and an expression for the method body. However, each method declaration includes an additional piece of information of the form `requires` Σ, where Σ stands for the *session environment* the said method requires for execution. The session environment Σ is a list of mappings of the form `ConvId` $\times$ `ChId` $\rightarrow$ `S`, where each pair of conversation and constituent channel is mapped to a session type. In other words we are *asserting* Σ as a precondition for the method. Since a method may involve communication actions over channels of conversations, by asserting Σ we ensure that these communication actions are over the conversation channels included in Σ and that they comply with the asserted session type.

3.2.4 Conversation Declaration

The representation of a conversation consists of a `ConvId` for the name of the conversation; a collection of roles of the form `RoleId` $\rightarrow$ `ClassId`, where `RoleId` stands for the name of the role and `ClassId` indicates its type; and finally a list of channels of the form `ChId` $\rightarrow$ `RoleId` $\times$ `RoleId` $\times$ `S`, with (`RoleId` $\times$ `RoleId`) denoting the roles *sending* and *receiving* over `ChId` respectively and `S` indicating the *session type* of the channel. Note that the channel name `ChId` is the endpoint accessed by the participants of the *sender* role. Those participating in the *receiver* role, can access this channel by using `dual(ChId)`, where

`dual(ch) = d(ch)` and `dual(d(ch)) = ch`.

3.2.5 Source Language Syntax

The syntax of the first three lines of expressions is as expected. In what follows we will explain the remaining expressions.

- `new ConId()`: Creates a new instance of the conversation of type `ConId`.
- `e.start()`: Results in initiation of the conversation pointed at by the receiver (`e`), provided that `e` evaluates to an instance of a conversation, i.e. of type `ConId`.
- `e.ch.send(e')`: Provided that the receiver (`e`) evaluates to an instance of a conversation, say `cv`, this expression results in sending the value derived from evaluating `e'` over the channel `ch` of `cv`.
- `x = e.ch.receive() in e'`: Provided that the receiver (`e`) evaluates to an instance of a conversation, say `cv`, this expression results in receiving a message over the channel `ch` of `cv`. Note that since when receiving a process needs to receive messages sent by *all* participants of the dual role, we have nested the receiving expression inside `e'` to avoid session interleaving. In other words, this statement acts as a while loop and constantly probes the channel for the next messgae to be received and substitutes x in expression `e'` with the received value. Once the process has received all messages as part of the current receiving stage, if the process attempts to receive more messages, it will be given a special message `EOM` to indicate the end of current receiving stage.
- `e.r.join() / e.r.leave()`: Results in the adding / removing of the currently active thread identifier to the list of participants in role `r` of the conversation instance derived from evaluating `e`.

3.2.6 Heap

This contains information about objects (class instances), conversation instances and process identifiers and we summarise it as follows.

- `ObjAddr` $\mapsto$ `object` : Denotes mappings from object identifiers to class instances (objects). An Object in turn keeps track of its type (`ClassId`) and the values associated with each of its fields.
- `ConvAddr` $\mapsto$ `conv` : Denotes mappings from conversation identifiers to conversation instances. A conversation retains information about its type (`ConId`); a list of its roles as well as the processes which participate in each role (`RoleId` $\rightarrow$ `ProcAddr`*); and finally the list of all channels

and the values communicated over each channel ($\texttt{ChId} \times \texttt{int} \rightarrow \texttt{value}^*$). Since communication is broadcast by default, at each stage of communication, *every* participant in the *sender* role sends a value over the channel and *each* member of the *receiver* role receives *all* communicated values. For instance, a mapping of the form $(\texttt{ch}, \texttt{i}) \rightarrow \{1, 2, 3\}$, is read as: "At the $\texttt{i}^{\texttt{th}}$ stage of communication, participants of the sender role communicated the values {1,2,3} over the channel ch." In a sense this mapping acts as a *noticeboard* of the channel organising all values communicated by the *senders* of the channel according to the communication *stage*.
Note that for each ($\texttt{ChId} \times \texttt{int} \rightarrow \texttt{value}^*$) mapping which represents the values sent from sender role ($\texttt{r}_1$) to receiver role ($\texttt{r}_2$), there is a *dual* mapping of the form ($\texttt{dual}(\texttt{ChId}) \times \texttt{int} \rightarrow \texttt{value}^*$), representing the values sent from ($\texttt{r}_2$) to ($\texttt{r}_1$).

Furthermore, there are no restrictions over the order in which values are sent at each communication stage. For instance, if the participants of the sender role are $\{\texttt{p}_1, \texttt{p}_2, ..., \texttt{p}_\texttt{n}\}$ and each $\texttt{p}_\texttt{i}$ is to communicate some value $\texttt{v}_\texttt{i}$ over channel ch at stage i, then $\overline{\texttt{v}}$ in the mapping $(\texttt{ch}, \texttt{i}) \rightarrow \overline{\texttt{v}}$ can be any permutation of $\{\texttt{v}_1, \texttt{v}_2, ..., \texttt{v}_\texttt{n}\}$.

- ($\texttt{ProcAddr} \times \texttt{ConvAddr} \times \texttt{ChId}$) $\rightarrow$ ($\texttt{int} \times \texttt{int} \times \texttt{int}$) The participants of each role can have access to multiple channels in a conversation and each process identifier can take part in several conversations. As a result, each process identifier can be communicating over multiple channels in different conversations. However, each channel is prescribed a distinct session type (S) and hence, a process participating in several conversations is expected to behave accordingly within each conversation. Furthermore, a participant of multiple roles might be at different stages of communication over each channel. For instance, assume that channel ch is defined as follows.

$$\texttt{channel ch r}_1\ \texttt{r}_2\ : !\texttt{C}_1.!\texttt{C}_2.?\texttt{C}_3$$

Now assume that process p is a participant in both $\texttt{r}_1$ and $\texttt{r}_2$. As a participant of role $\texttt{r}_1$, he sends a value of type $\texttt{C}_1$ at *sending stage 1*, followed by a value of type $\texttt{C}_2$ at *sending stage 2* and finally expects to receive a value of type $\texttt{C}_3$ at *receiving stage 1*. As a participant of role $\texttt{r}_2$ on the other hand, he expects to receive a value of type $\texttt{C}_1$ at *receiving stage 1*, followed by a value of type $\texttt{C}_2$ at *receiving stage 2* and finally sends a value of type $\texttt{C}_3$ at *sending stage 1*. Hence,

process `p` can be at different *stages* of sending and receiving within each role. For this reason, every $(\pi, \texttt{cv}, \texttt{ch})$ tuple in heap is mapped to a `(i,j,k)` tuple, where `i` and `j` refer to *sending stage* and *receiving stage* respectively and `k` indicates the *receiving index*. As discussed above, when receiving over a channel, a process must receive *all* values sent by the senders. Since the receiving action over a channel might be interleaved by other communication actions over different channels, a process must remember the index of the last value it received to avoid receiving duplicate messages.

3.2.7 Thread

Each thread is a pair of the form $(\texttt{e}, \texttt{ProcAddr})$, where `e` refers to the expression the thread is to execute and `ProcAddr` denotes the identity of the process. The *Roles* language is designed as a multi-threaded concurrent language, hence we allow for parallel composition of threads, denoted by `P | P`.

Note that we require each thread to carry its identity, since participants of roles are simply processes and roles keep track of their members by recording their identifiers. Therefore, when a `join` or `leave` expression is executed, it is the identifier of the active thread that is added/removed from the relevant role.

3.2.8 Naming Conventions

We will use the following naming conventions for various identifiers throughout the remainder of this report.

Source Entities

$\texttt{f} \in \texttt{FldId}$	for field identifiers.
$\texttt{m} \in \texttt{MethId}$	for method identifiers.
$\texttt{C} \in \texttt{ClassId}$	for class identifiers.
$\texttt{CV} \in \texttt{ConId}$	for conversation identifiers.
$\texttt{r} \in \texttt{RoleId}$	for role identifiers.
$\texttt{ch}, \texttt{d}(\texttt{ch}) \in \texttt{ChId}$	for channel identifiers.
$\texttt{o} \in \texttt{ObjId}$	for object identifiers.
$\texttt{cv} \in \texttt{CVId}$	for conversation instance identifiers.
$\texttt{s} \in \texttt{S}$	for session types.

Runtime Entities

$\iota \in \texttt{ObjAddr}$	for the address of an object.
$\kappa \in \texttt{ConvAddr}$	for the address of a conversation instance.
$\pi \in \texttt{ProcAddr}$	for process identifiers.
$\texttt{v} \in \texttt{value}$	for values.
$\chi \in \texttt{heap}$	for heaps.

3.3 Auxiliary Functions

In the sections to follow, we refer to the following table lookup functions to extract information from the source code or various entries in the heap.

dual(ch) = d(ch)

dual(d(ch)) = ch

FD(Prog, C, f) = Prog(C)$\downarrow_2$(f)

$$\text{F(Prog, C, f)} = \begin{cases} \text{FD(Prog, C, f)} & \text{if FD(Prog, C, f)} \neq \mathit{Udf} \\ \text{F(Prog, Prog(C)} \downarrow_1 \text{, f)} & \text{otherwise} \end{cases}$$

FS(Prog, C) = {f | F(Prog, C, f) $\neq \mathit{Udf}$}

MD(Prog, C, m) = Prog(C)$\downarrow_3$(m)

$$\text{M(Prog, C, m)} = \begin{cases} \text{MD(Prog, C, m)} & \text{if MD(Prog, C, m)} \neq \mathit{Udf} \\ \text{M(Prog, Prog(C)} \downarrow_1 \text{, m)} & \text{otherwise} \end{cases}$$

MS(Prog, C)= {m | M(Prog, C, m) $\neq \mathit{Udf}$}

$$\text{mType (Prog, C, m)} = \begin{cases} (\overline{\text{t}} \rightarrow \text{t}), \Sigma & \text{if M(Prog, C, m)} = \text{t m } (\overline{\text{t x}}) \text{ requires } \Sigma \ \{\overline{\text{e}}\} \\ \bot & \text{otherwise} \end{cases}$$

$$\text{requires(Prog, C, m)} = \begin{cases} \Sigma & \text{if mType(Prog, C, m)} = (\overline{\text{t}} \rightarrow \text{t}), \Sigma \\ \bot & \text{otherwise} \end{cases}$$

$$\text{mBody (Prog, C, m)} = \begin{cases} \overline{\text{e}} & \text{if M(Prog, C, m)} = \text{t m } (\overline{\text{t x}}) \text{ requires } \Sigma \ \{\overline{\text{e}}\} \\ \bot & \text{otherwise} \end{cases}$$

$$\text{mArgs (Prog, C, m)} = \begin{cases} \overline{\text{x}} & \text{if M(Prog, C, m)} = \text{t m } (\overline{\text{t x}}) \text{ requires } \Sigma \ \{\overline{\text{e}}\} \\ \bot & \text{otherwise} \end{cases}$$

$$\text{CH(Prog, CV, ch)} = \text{Prog(CV)}{\downarrow_2}\text{(ch)}$$

$$\text{CHS(Prog, CV)} = \{\text{ch} \mid \text{CH(Prog, CV, ch)} \neq \mathit{Udf}\}$$

$$\begin{aligned}\text{chanIDs (Prog, CV)} = \quad & \{\text{ch} \mid \text{CH(Prog, CV, ch)} \neq \mathit{Udf}\} \\ & \cup \{\text{dual(ch)} \mid \text{CH(Prog, CV, ch)} \neq \mathit{Udf}\}\end{aligned}$$

$$\text{sender (Prog, CV, ch)} = \begin{cases} \text{r}_1 & \text{if CH(Prog, CV, ch)} = (\text{r}_1, \text{r}_2, \text{S}) \\ \text{r}_2 & \text{if CH(Prog, CV, dual(ch))} = (\text{r}_1, \text{r}_2, \text{S}) \\ \bot & \text{otherwise} \end{cases}$$

$$\text{receiver (Prog, CV, ch)} = \begin{cases} \text{r}_2 & \text{if CH(Prog, CV, ch)} = (\text{r}_1, \text{r}_2, \text{S}) \\ \text{r}_1 & \text{if CH(Prog, CV, dual(ch))} = (\text{r}_1, \text{r}_2, \text{S}) \\ \bot & \text{otherwise} \end{cases}$$

$$\text{session (Prog, CV, ch)} = \begin{cases} \text{S} & \text{if CH(Prog, CV, ch)} = (\text{r}_1, \text{r}_2, \text{S}) \\ \text{dual(S)} & \text{if CH(Prog, CV, dual(ch))} = (\text{r}_1, \text{r}_2, \text{S}) \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{aligned}&\text{dual(!C.s)} = \text{?C.dual(s)} \\ &\text{dual(?C.s)} = \text{!C.dual(s)}\end{aligned}$$

$$\begin{aligned}\text{role\_channels(Prog, CV, r)} = \quad & \{\text{ch} \mid \text{sender(Prog, CV, ch)} = \text{r}\} \\ & \cup \{\text{dual(ch)} \mid \text{receiver(Prog, CV, ch)} = \text{r}\}\end{aligned}$$

$$\text{RS(Prog, CV)} = (\text{Prog(CV)}) \downarrow_1$$

$$\text{parts(Prog, } \chi\text{, } \kappa\text{, r)} = \{\pi \mid \pi \in \chi(\kappa) \downarrow_2 (\text{r})\}$$

$$\begin{aligned}\text{senders(Prog, } \chi\text{, } \kappa\text{, ch)} = \{\pi \mid \quad & \chi(\kappa) \downarrow_1 = \text{CV} \\ & \wedge\ \text{sender(Prog, CV, ch)} = \text{r} \\ & \wedge\ \pi \in \chi(\kappa) \downarrow_2 (\text{r}) \qquad \}\end{aligned}$$

$$\begin{aligned}\text{receivers(Prog, } \chi\text{, } \kappa\text{, ch)} = \{\pi \mid \quad & \chi(\kappa) \downarrow_1 = \text{CV} \\ & \wedge\ \text{receiver(Prog, CV, ch)} = \text{r} \\ & \wedge\ \pi \in \chi(\kappa) \downarrow_2 (\text{r}) \qquad \}\end{aligned}$$

$$\text{out\_T}\ (\text{S, n}) = \begin{cases} \text{T(S', n-1)} & \text{if S = !C.S'} \ \wedge \ \text{n} > 0 \\ \text{C} & \text{if S = !C.S'} \ \wedge \ \text{n} = 0 \\ \text{T(S', n)} & \text{if S = ?C.S'} \ \wedge \ \text{n} \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

$$\text{subS(S, w, r)} = \begin{cases} \text{subS(S', w-1, r)} & \text{if S = !C.S'} \ \wedge \ \text{w} > 0 \\ \text{subS(S', w, r-1)} & \text{if S = ?C.S'} \ \wedge \ \text{r} > 0 \\ \text{S} & \text{if w} = 0 \ \wedge \ \text{r} = 0 \\ \bot & \text{otherwise} \end{cases}$$

$$\text{started}(\chi, \text{cv}) = \begin{cases} \text{true} & \text{if } \chi(\text{cv}) \downarrow_3 \neq \epsilon \\ \text{false} & \text{if } \chi(\text{cv}) \downarrow_3 = \epsilon \\ \bot & \text{otherwise} \end{cases}$$

3.4 Operational Semantics

In this section we will introduce the operational semantics of *Roles* inspired by standard small step reduction of [18]. Prior to discussing the operational semantics rules, we first list the evaluation contexts below.

3.4.1 Evaluation Contexts

```
Ctxt ::= - | -;e
         | -.f | -.f:=e | o.f := -
         | -.m(e̅) | o.m(v̅, -, e̅)
         | -.ch.send(e) | cv.ch.send(-)
         | x = cv.ch.receive() in -
         | -.start()
         | -.r.join() | -.r.leave()
```

3.4.2 Rewriting Rules

The rewriting rules of *Roles* are judgements of the form:

$$(\mathsf{e}, \pi)|\overline{\mathsf{P}},\ \chi \ \longrightarrow\ (\mathsf{e}', \pi)|\overline{\mathsf{P}}|\overline{\mathsf{P}'}, \chi'$$

$(\mathsf{e}, \pi)|\overline{\mathsf{P}}$ refers to the list of present threads with (e, π) being the currently active thread, that is the one that undergoes reduction. π stands for the process identifier and `e` denotes the expression to execute. Given *initial* heap χ, the (e, π) thread rewrites to (e', π) with χ' as the *final* heap where the effects of evaluating e have been reflected in χ'. All other processes in $\overline{\mathsf{P}}$ remain unchanged and are carried forward. Furthermore, evaluating (e, π) may result in spawning new processes ($\overline{\mathsf{P}'}$), hence growing the list of present threads. This has been reflected in the semantics of the **Spawn** and **Start** rules.

We will now present the operational semantics of *Roles* followed by an explanation of each rule. We have included the operational semantics of *Roles* in appendix B, for reference.

Process

$$\frac{\mathsf{P}_1,\ \chi \rightsquigarrow\ \overline{\mathsf{P}_2},\ \chi'}{\overline{\mathsf{P}_a} \mid \mathsf{P}_1 \mid \overline{\mathsf{P}_b}\ ,\ \chi \rightsquigarrow \overline{\mathsf{P}_a} \mid \overline{\mathsf{P}_2} \mid \overline{\mathsf{P}_b}\ ,\ \chi'}$$

Since our operational semantics is based on the reduction of a single thread at a time, this rule gives the operational semantics of the overall system consisting of multiple threads. This rule asserts that if a single process $\mathtt{P_1}$ can be rewritten to $\overline{\mathtt{P_2}}$, then a system consisting of $(\overline{\mathtt{P_a}}|\mathtt{P_1}|\overline{\mathtt{P_b}})$ processes can be rewritten to $(\overline{\mathtt{P_a}}|\overline{\mathtt{P_2}}|\overline{\mathtt{P_b}})$.

Context

$$\frac{(\mathsf{e},\pi),\chi \rightsquigarrow (\mathsf{e}',\pi)|\overline{\mathsf{P}},\chi'}{(\mathsf{Ctxt}[\mathsf{e}],\pi),\chi \rightsquigarrow (\mathsf{Ctxt}[\mathsf{e}'],\pi)|\overline{\mathsf{P}},\chi'}$$

Since our operational semantics is based on small step reductions, this rule gives the operational semantics of an expression in an evaluation contexts. The evaluation contexts of *Roles* are listed in section 3.4.1.

Spawn

$$\frac{\pi' \notin \chi \qquad \chi' = \chi[(\pi,\mathsf{null},\mathsf{null}) \mapsto (0,0,0)]}{(\mathsf{spawn}\{\mathsf{e}\},\pi),\chi \rightsquigarrow (\mathsf{null},\pi)|(\mathsf{e},\pi'),\chi'}$$

This rule is used for spawning a new thread. The new thread is given a fresh address in the heap. The newly spawned thread then starts executing in parallel with the present thread(s) but since there is nothing to execute at this stage, a `null` expression is associated with this thread.

Fld

$$\frac{\chi(\iota) = (\mathsf{C}, \overline{\mathsf{f} : \mathsf{v}})}{(\iota.\mathsf{f_i},\ \pi), \chi \rightsquigarrow (\mathsf{v_i}, \pi), \chi}$$

This rule is used to access a field of an object in the heap in the standard way. First, the heap is scanned for the address of the object (ι) and once the receiver object is found, the value associated with the desired field is looked up in the field table and returned.

FldAss

$$\frac{\chi' = \chi[\iota \mapsto \chi(\iota)[\mathsf{f} \mapsto \mathsf{v}]]}{(\iota.\mathsf{f} := \mathsf{v}\ ,\ \pi), \chi \rightsquigarrow (\mathsf{v}, \pi), \chi'}$$

This rule is used to assign a value to a field of an object. First, we search the heap for the address of the receiver object and once it is found the value of the target field is updated in the field table of the object with the given value.

NewC

$$\frac{\mathsf{FS}(\mathsf{C}) = \overline{\mathsf{C}\ \mathsf{f}} \qquad \iota \notin \chi}{(\mathsf{new}\ \mathsf{C}\ ,\ \pi), \chi \rightsquigarrow (\iota, \pi), \chi[\iota \mapsto (\mathsf{C}, \overline{\mathsf{f} : \mathsf{null}})]}$$

This is used to create a new instance of a class. We first allocate a fresh address (one that is not already in use in the heap) to the new instance and then initialise all its fields with the default `null` value.

Meth

$$\frac{\chi(\iota) = (\mathsf{C}, \overline{\mathsf{f} : \mathsf{v}}) \quad \mathsf{mBody}(\mathsf{Prog}, \mathsf{C}, \mathsf{m}) = \mathsf{e}}{(\iota.\mathsf{m}(\overline{\mathsf{v}})\ ,\ \pi), \chi \rightsquigarrow (\mathsf{e}[\iota/\mathsf{this}][\overline{\mathsf{v}}/\overline{\mathsf{x}}]\ ,\ \pi), \chi}$$

This rule is used for method calls. The semantics of this rule is standard up to session types. Each method signature includes an additional piece of information which acts as the *precondition* of the method. This precondition Σ is a session environment and asserts the behaviour it expects from each channel of a conversation. If the session types of channels specified in Σ *agree* with the channel session types of the current process, then the precondition is fulfilled and method body e can be executed.

NewCV

$$\frac{\mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) = \overline{\mathsf{r}} \quad \kappa \notin \chi \quad \chi' = \chi[\kappa \mapsto (\mathsf{CV}, \overline{\mathsf{r} : \epsilon}, \epsilon)]}{(\mathsf{new}\ \ \mathsf{CV}, \pi)\ , \chi \rightsquigarrow\ (\kappa, \pi), \chi'}$$

This rule is used to create a new instance of a conversation. We first allocate a fresh address (one that is not already in use in the heap) to the new instance and then initialise all its roles with empty lists since initially none of the roles have any participants. Note that the channels of the conversation are not initialised, since we don't want the channels to be used for communication *prior* to the start of the conversation. Hence, we leave the channel table of the conversation empty when it is created.

Join

$$\frac{\chi(\kappa) \downarrow_3 = \epsilon \quad \chi' = \chi[\ (\kappa, \mathsf{r}) + = \pi]}{(\kappa.\mathsf{r.join}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi), \chi'}$$

This rule is used to add a new participant to a role of a conversation. However, processes can only join a conversation if it has not already started. We therefore first check the channel table of the said conversation in the heap; an empty channel table indicates that the conversation has not started yet and it is permissible for new proceses to join any of its roles. We then add current thread's identifier to the participant list of the relevant role in the heap.

Leave

$$\frac{\pi \in \chi(\kappa) \downarrow_2 (\mathsf{r}) \qquad \chi' = \chi[\ (\kappa, \mathsf{r}) - = \pi]}{(\kappa.\mathsf{r.leave}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi), \chi'}$$

This role is analogous to the join rule described above. We use this rule to remove a participant from a role of a conversation. However, unlike the semantics of the join rule, a process is allowed to leave a conversation at any time even if the conversation has already started. We first check the participant list of the relevant role and if the currently active thread is a member of the role in question we remove it from its participant list.

Start

$$\frac{\begin{array}{l}\chi(\kappa) = (\mathsf{CV},\ \overline{\mathsf{r} : \overline{\pi}},\ \epsilon) \\ \chi'' = \chi[\kappa \mapsto (\mathsf{CV},\ \overline{\mathsf{r} : \overline{\pi}},\ \mathsf{ChInit})] \\ \qquad \forall\, \mathsf{ch} \in \mathsf{ChanIDs}(\mathsf{Prog}, \mathsf{CV}) :\ \ \mathsf{ChInit}(\mathsf{ch}, 0) = \epsilon \\ \qquad\qquad\qquad\qquad \forall\, \mathsf{i} \neq 0 :\ \mathsf{ChInit}(\mathsf{ch}, \mathsf{i}) = \bot \\ \forall\, \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) :\ \forall\, \pi \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) :\ \forall\, \mathsf{ch} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) : \\ \qquad \chi'(\pi, \kappa, \mathsf{ch}) = (0, 0, 0) \\ \qquad \chi'(\mathsf{z}) = \chi''(\mathsf{z}) \quad \text{if } \mathsf{z} \notin (\pi, \kappa, \mathsf{ch})\end{array}}{(\kappa.\mathsf{start}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi),\ \chi'}$$

This rule is used to initiate a conversation that has not yet started. Upon initiation of the conversation we need to initialise its channel table, since prior to the start of the conversation no communication was allowed on any of its channels. We now allocate a message table (a noticeboard) for each of its channels and their *duals* where messages can be dropped or retrieved by those communicating on the channels.

Furthermore, for each of the participants in the roles, we need to add *indexing information* per channel. Since each process can be a member of multiple roles and hence communicating over several channels, it needs to remember which sending/receiving round it is at; which row in the channel table to drop or retrieve his messages from. Finally, since when a process is receiving a message it needs to receive it from *all* participants of the dual role, in order to avoid receiving duplicate messages it needs to remember the index of the last message he retrieved in the current receiving round. Note that all three indices are initialised with zero.

Message Tables Explained

In order to clarify the semantics of the next rules, we first try to give the reader a better understanding of message tables for channels in the heap. Suppose roles $\mathtt{r}_1$ and $\mathtt{r}_2$ have `n` and `m` participants respectively. Now assume channel `ch` is declared between $\mathtt{r}_1$ and $\mathtt{r}_2$ and is defined as thus:

$$\texttt{channel ch}\ \mathtt{r}_1\ \mathtt{r}_2\ \texttt{:!Int.?Char.!Bool.?Int}$$

There are two message tables (noticeboards) for channel `ch`. One for the `ch` endpoint where the participants of the $\mathtt{r}_1$ role drop their messages, and another message table for the `d(ch)` endpoint where the participants of the $\mathtt{r}_2$ role drop their messages.

Note that each message table is associated with an endpoint and is where the participants of the role using the endpoint *send* their messages. For instance, whenever the participants of the $\mathtt{r}_1$ role *send* a message, it is dropped in the message table of the `ch` endpoint, the endpoint used by the members of the $\mathtt{r}_1$ role.

On the other hand, whenever participants of a role attempt to *receive* a message over a channel, they use the *dual* of the endpoint they use for sending. For instance, when participants of the $\mathtt{r}_1$ role try to receive a message over channel `ch`, they retrieve it from the message table of the `d(ch)` endpoint.

Similarly, the partcipants of the role `r2` use the message table of the `d(ch)` endpoint to drop their messages and when receiving, they retrieve their messages from the message table of the `ch` endpoint.

Communication is organised into several *rounds* or *stages* of sending and receiving, since for each sending (receiving) action several values are communicated depending on the cardinality of roles. In the example above, the session type of the `ch` channel consists of two sending rounds and two reciving rounds.

The message table for the channel `ch` can be visualised in the heap as in Figure 3.2. As mentioned above, this message table is used by the members of the `r1` role to drop their messages. As part of the *first sending stage*, the `n` participants of role `r1` send values of type `Int` which is reflected in the *first row* of the table.

Next, they move on to receiving `m` messages of type `Char` from participants of role `r2` at the *first receiving stage*. However, as discussed above, when receiving, participants retrieve their messages from the message box of the dual endpoint - in this case `d(ch)`. Therefore, members of the `r1` role inspect the *first* row of the message table associated with the `d(ch)` endpoint for `m` messages to be received from the participants of the `r2` role.

Subsequently, each participant then sends a message of type `Bool` as part of the *second sending stage* as shown in the *second* column of the table in Figure 3.2. Finally, they retrieve `m` messages of type `Int` from the message table of the `d(ch)` endpoint.

sending stage \ participants	1	2	...	n
1	int_1	int_2	...	int_n
2	$bool_1$	$bool_2$	...	$bool_n$
...	...	...	...	...

Figure 3.2: The message table of the `ch` endpoint in the heap.

It follows that the session type of the `d(ch)` channel is:

`?Int.!Char.?Bool.!Int`

The message table of channel `d(ch)` is as shown in Figure 3.3 where the participants of role $\mathtt{r}_2$ show the dual behaviour of those in $\mathtt{r}_1$.

sending stage \ participants	1	2	...	m
1	$char_1$	$char_2$	...	$char_m$
2	int_1	int_2	...	int_m
...	...	...	...	...

Figure 3.3: The message table of the `d(ch)` endpoint in the heap.

Note that *sending stage* refers to the row of the table at which a participant is to drop his message and a *receiving stage* is the row of the message table a participant is to retrieve its message from. Furthermore, given the receiving stage, *receiving index* refers to the index of the message that is to be retrieved at the current receiving stage. In other words, the receiving stage together with the receiving index provide us with a way of locating the exact position of the message that is to be received next.

Having now explored the message tables, we will now continue with the remaining reduction rules.

Send

$$\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k})$$
$$\chi_1 \;=\; \chi[(\pi, \kappa, \mathsf{ch}) \;\mapsto\; (\mathsf{i}+1, \mathsf{j}, \mathsf{k})]$$
$$\chi_2 \;=\; \chi_1(\kappa) \downarrow_3 [(\mathsf{ch}, \mathsf{i}) \mapsto \chi_1(\kappa) \downarrow_3 (\mathsf{ch}, \mathsf{i}) +\!+ [\mathsf{v}]]$$

$$(\kappa.\mathsf{ch}.\mathsf{send}(\mathsf{v}),\ \pi),\ \chi \rightsquigarrow (\mathsf{null}, \pi),\ \chi_2$$

This rule is used for a process to send a message over a channel of a conversation. First, the process's indexing information - the sending round in particular - for the appropriate channel of the said conversation is retrieved. It then updates the message table of the channel by appending its message to the row indexed by the sending stage. Finally, since it has finished sending

as part of the current sending stage, its sending stage index is incremented.

The following rules are used for a process to receive a message over a channel of a conversation.

Receive1

$$\frac{\begin{array}{l}\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k}) \\ \mathsf{k} = \sharp\ \mathsf{senders}(\mathsf{Prog}, \chi, \kappa,\ \mathsf{ch}) \\ \chi' \ =\ \chi[(\pi, \kappa, \mathsf{ch}) \ \mapsto\ (\mathsf{i}, \mathsf{j}+1, 0)]\end{array}}{(\mathsf{x} = \kappa.\mathsf{ch.receive}()\ \mathsf{in}\ \mathsf{e},\ \pi), \chi \rightsquigarrow (\mathsf{EOM}, \pi), \chi'}$$

In the case that the process has received *all* messages sent by the members of the dual role, that is to say the *receiving index* is equal to the number of participants sending over the channel minus one (since receiving index starts at zero), the process will receive a special message, namely EOM. Since it has reached the end of current receiving stage, this index is incremented and its receiving index is reset to zero.

Receive2

$$\frac{\begin{array}{l}\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k}) \\ \mathsf{k} < \sharp\ \mathsf{senders}(\mathsf{Prog}, \chi, \kappa,\ \mathsf{ch}) \\ \chi(\kappa) \downarrow_3 (\mathsf{dual}(\mathsf{ch}), \mathsf{j})[\mathsf{k}] = \mathsf{v} \\ \chi' \ =\ \chi[(\pi, \kappa, \mathsf{ch}) \mapsto (\mathsf{i}, \mathsf{j}, \mathsf{k}+1)]\end{array}}{(\mathsf{x} = \kappa.\mathsf{ch.receive}()\ \mathsf{in}\ \mathsf{e},\ \pi), \chi \rightsquigarrow (\ (\mathsf{e}[\mathsf{v}/\mathsf{x}];\ \mathsf{x}\ =\ \kappa.\mathsf{ch.receive}()\ \mathsf{in}\ \mathsf{e})\ , \pi), \chi'}$$

If the premise of the previous rule doesn't hold and the process has not yet received messages from all participants of the dual role, the message addressed by the receiving stage and receiving index in the message table is retrieved. The receiving index is then incremented to point to the next message to be received.

Definition (Participant Joining and Leaving)

The following shorthand have been used in the semantics of the `Join` and `Leave` reduction rules and are defined as thus.

$\chi[\ (\kappa, \mathsf{r})+ = \pi]$ is an abbreviation of:
$\chi[\kappa \mapsto (\ \chi(\kappa)\downarrow_1\ ,\ \chi(\kappa)\downarrow_2 [\mathsf{r} \mapsto \chi(\kappa)\downarrow_2 (\mathsf{r}) \cup \{\pi\}],\ \chi(\kappa)\downarrow_3\ )\ ]$

$\chi[\ (\kappa, \mathsf{r})- = \pi]$ is an abbreviation of:
$\chi[\kappa \mapsto (\ \chi(\kappa)\downarrow_1\ ,\ \chi(\kappa)\downarrow_2 [\mathsf{r} \mapsto \chi(\kappa)\downarrow_2 (\mathsf{r}) \setminus \{\pi\}],\ \chi(\kappa)\downarrow_3\ )\ ]$

Sending Example

Assume that the process π, is executing expression `e` of the form:

$$\kappa.\mathsf{ch}.\mathsf{send}(7)$$

and the indexing information for π is thus:

$$\chi(\pi, \kappa, \mathsf{ch}) = (15, 17, 0)$$

Finally, assume that the message table of the `ch` endpoint in the heap χ looks as in Figure 3.4.

Once expression `e` is executed, heap χ will be modified to produce heap χ' as stated in the reduction rule for send. We then have:

$$\chi'(\pi, \kappa, \mathsf{ch}) = (16, 17, 0)$$

The message table of the `ch` endpoint in the heap χ' then looks as in Figure 3.5.

sending stage \ participants	1	2	...	...	...	m
1	...	...	...	...	...	...
2	...	...	...	...	...	...
...	...	...	...	...	...	...
15	int_1	...	int_k	ϵ	...	ϵ
...	...	...	...	...	...	...

Figure 3.4: The message table of the `ch` endpoint in the heap, *before* executing expression `e`.

sending stage \ participants	1	2	...	...	...	m
1	...	...	...	...	...	...
2	...	...	...	...	...	...
...	...	...	...	...	...	...
15	int_1	...	int_k	7	...	ϵ
...	...	...	...	...	...	...

Figure 3.5: The message table of the `ch` endpoint in the heap, *after* executing expression `e`.

Receiving Example

Assume that the process π, is executing the expression `e` of the following form where `i` is initially set to zero.

$$\mathsf{x} \;=\; \kappa.\mathsf{ch.receive}()\ \mathsf{in}\ \{\ \mathsf{a[i]} \;=\; \mathsf{x};\ \mathsf{i}++\ \}$$

and the indexing information for π is as thus:

$$\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{17}, \mathsf{10}, \mathsf{0})$$

Finally, assume that there are `n` values to be received and that the message table of the `d(ch)` endpoint in the heap χ looks as in Figure 3.7.

Once expression `e` is executed, the message box of the `d(ch)` endpoint in the heap χ will be repeatedly inspected for subsequent values to be received and the indexing information of the π in the heap changes after receiving each message. Finally where there no longer any messages to be received as part of the current receivng round, an `EOM` message is returned and the indices are modified accordingly. This procedure is visualised in Figure 3.8.

Finally, once the execution of this instruction is complete, the values in array `a` are as in Figure 3.6. Note that when receiving, the message box remains intact and the received values are *not* removed from it. Therefore, the message box of the `d(ch)` endpoint *after* execution is the same as Figure 3.7.

---

index	0	1	...	m-1
value	int_1	int_2	...	int_m

Figure 3.6: Values of entries in array `a` after the execution of expression `e`.

---

sending stage \ participants	1	2	...	...	...	n
1	...	...	...	...	...	...
2	...	...	...	...	...	...
...	...	...	...	...	...	...
10	int_1	...	...	...	...	int_n
...	...	...	...	...	...	...

Figure 3.7: The message table of the `d(ch)` endpoint in the heap, *before* and *after* executing expression `e`.

Heap (before)	**Indexing (before)**	**Value Received**	**Heap (after)**	**Indexing (after)**
χ	(17, 10, 0)	int_1	χ_1	(17, 10, 1)
χ_1	(17, 10, 1)	int_2	χ_2	(17, 10, 2)
χ_2	(17, 10, 2)	int_3	χ_3	(17, 10, 3)
...	...	...	...	...
χ_{n-1}	(17, 10, n-1)	int_n	χ_n	(17, 10, n)
χ_n	(17, 10, n)	`EOM`	χ'	(17, 11, 0)

Figure 3.8: Iterations for receiving `n` messages from the message box of `d(ch)` endpoint.

3.5 Type System

3.5.1 Subclassing

The judgement $\mathsf{Prog} \vdash \mathsf{C} \sqsubseteq \mathsf{C}'$ asserts that class C is a subclass of class C' and is defined as follows as defined in [20].

$$\frac{}{\vdash \mathsf{Object} \sqsubseteq \mathsf{Object}}$$

$$\frac{\mathsf{Prog}(\mathsf{C}) \downarrow_1 = \mathsf{C}'}{\mathsf{C} \sqsubseteq \mathsf{C}' \qquad \mathsf{C} \sqsubseteq \mathsf{C}}$$

$$\frac{\mathsf{C} \sqsubseteq \mathsf{C}' \qquad \mathsf{C}' \sqsubseteq \mathsf{C}''}{\mathsf{C} \sqsubseteq \mathsf{C}''}$$

3.5.2 Subtyping

The subtyping relationship ($\leq$) as defined in [20] is the projection of subclassing relationship onto types as defined below.

$$\frac{\mathsf{Prog} \vdash \mathsf{C} \sqsubseteq \mathsf{C}'}{\mathsf{Prog} \vdash \mathsf{C} \leq \mathsf{C}'}$$

$$\frac{\mathsf{Prog} \vdash \mathsf{t} \diamond_{\mathsf{t}}}{\mathsf{Prog} \vdash \mathsf{t} \leq \mathsf{t}}$$

3.5.3 Acyclic Class Hierarchy

The judgement $\vdash \mathsf{Prog}\diamond_{\mathsf{a}}$ asserts that the class hierarchy in program Prog is acyclic and is defined as follows.

$$\frac{\forall \mathsf{C}, \mathsf{C}' : \mathsf{Prog} \vdash \mathsf{C} \sqsubseteq \mathsf{C}' \text{ and } \mathsf{Prog} \vdash \mathsf{C}' \sqsubseteq \mathsf{C} \longrightarrow \mathsf{C} = \mathsf{C}' \quad \mathsf{Prog}(\mathsf{C}) \downarrow_1 = \mathsf{C}' \longrightarrow \mathsf{C} \neq \mathsf{C}'}{\vdash \mathsf{Prog}\diamond_{\mathsf{a}}}$$

Type Judgements

The judgement $\vdash \mathsf{C}\diamond_{\mathsf{c}}$, asserts that C is a class, $\vdash \mathsf{CV}\diamond_{\mathsf{cv}}$, asserts that CV is a conversation and $\vdash \mathsf{t}\diamond_{\mathsf{t}}$ asserts that t is a type.

$$\frac{\mathsf{C} \in \mathsf{Prog}}{\vdash \mathsf{C}\diamond_{\mathsf{c}} \quad \vdash \mathsf{C}\diamond_{\mathsf{t}}}$$

$$\frac{\mathsf{CV} \in \mathsf{Prog}}{\vdash \mathsf{CV}\diamond_{\mathsf{cv}} \quad \vdash \mathsf{CV}\diamond_{\mathsf{t}}}$$

3.5.4 Static Typing judgement

Typing an expression statically is a judgement of the form:

$$\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma'$$

That is to say, in the context of *environment* Γ, and the *initial session environment* Σ, the expression e, has type t while it *consumes* the initial session environment (Σ) and returns the *final session environment* Σ'. In other words, Σ' reflects the changes applied to the channel sessions initially defined in Σ through communication instructions **send** and **receive**.

3.5.5 Γ Environment

The environment Γ maps this to a class type and method arguments (x) to types (a class or a conversation). Thus, type environments are defined by the following syntax.

$$\Gamma \ ::= \emptyset \ \mid \ \Gamma, \mathsf{this} : \mathsf{ClassId} \ \mid \ \Gamma, \mathsf{x} : \mathsf{t}$$

3.5.6 Session Environment

The session environment Σ maps each conversation instance and its channels to session types and is defined by the following syntax.

$$\begin{array}{rl} & \Sigma ::= (\mathsf{CVId} \times \mathsf{ChId}) \rightarrow \mathsf{S} \\ \text{where} & \quad \mathsf{S} ::= ?(\mathsf{C}).\mathsf{S} \ \mid \ !(\mathsf{C}).\mathsf{S} \end{array}$$

The judgement $\mathsf{s}\diamond_{\mathsf{s}}$ asserts that s is a session.

$$\frac{\mathsf{s} = \dagger \mathsf{C}.\mathsf{s}' \quad \mathsf{C} \in \mathsf{Prog} \quad \mathsf{Prog} \vdash \mathsf{s}'\diamond_{\mathsf{s}}}{\mathsf{Prog} \vdash \mathsf{s}\diamond_{\mathsf{s}}}$$

where $\dagger$ refers to communication actions namely, ! and ?.

3.5.7 Session Environment Algebra

In this part we define a number of operations on session environments, including various orderings and subtraction.

Definition (Prefix Session Ordering)

$\Sigma \preceq_{pre} \Sigma'$ is read as session environment Σ is a *prefix* to session environment Σ' and is defined as follows. These operations are used when specifying the typing rules of the *Roles* language in later sections.

$$\Sigma \preceq_{\mathsf{pre}} \Sigma' \quad \text{iff} \quad \forall\, (\mathsf{cv}, \mathsf{ch}) \in \mathsf{dom}(\Sigma) :\ \Sigma(\mathsf{cv}, \mathsf{ch}) \ \preceq_{\mathsf{pre}}\ \Sigma'(\mathsf{cv}, \mathsf{ch})$$

where prefix ordering for sessions ($\mathsf{s} \preceq_{\mathsf{pre}} \mathsf{s}'$) is as defined below.

$$\mathsf{s}_1 \preceq_{\mathsf{pre}} \mathsf{s}_2 \quad \text{iff} \quad \mathsf{s}_2 = \mathsf{s}_1.\mathsf{s}$$

Definition (Suffix Session Ordering)

This is the mirror of the previous definition. We read $\Sigma \preceq_{suf} \Sigma'$ as session environment Σ is a *suffix* to session environment Σ' and is defined as follows.

$$\Sigma \preceq_{\mathsf{suf}} \Sigma' \quad \text{iff} \quad \forall\, (\mathsf{cv}, \mathsf{ch}) \in \mathsf{dom}(\Sigma) :\ \Sigma(\mathsf{cv}, \mathsf{ch}) \ \preceq_{\mathsf{suf}}\ \Sigma'(\mathsf{cv}, \mathsf{ch})$$

where suffix ordering for sessions ($\mathsf{s} \preceq_{\mathsf{suf}} \mathsf{s}'$) is as defined below.

$$\mathsf{s}_1 \preceq_{\mathsf{suf}} \mathsf{s}_2 \quad \text{iff} \quad \mathsf{s}_2 = \mathsf{s}.\mathsf{s}_1$$

Definition (Session Subtraction)

Session environment subtraction written as $\Sigma_1 - \Sigma_2$ is given below.

$$\Sigma_1 - \Sigma_2 = \begin{cases} \Sigma & \text{if } \Sigma_2 \preceq_{pre} \Sigma_1 \\ \bot & \text{otherwise} \end{cases}$$

where Σ is defined as follows.

$$\Sigma(\text{cv, ch}) = \begin{cases} Udf & \text{if } \Sigma_1(\text{cv, ch}) = Udf \\ \text{s}_1 & \text{if } \Sigma_1(\text{cv, ch}) = \text{s}_1 \ \wedge\ (\ \Sigma_2(\text{cv, ch}){=}Udf \ \vee\ \Sigma_2(\text{cv, ch}){=}\epsilon\ ) \\ \text{s}_1 - \text{s}_2 & \text{if } \Sigma_1(\text{cv, ch}) = \text{s}_1 \ \wedge\ \Sigma_2(\text{cv, ch}) = \text{s}_2 \end{cases}$$

Session type subtraction ($\mathsf{s}_1 - \mathsf{s}_2$) is defined by:

$$\mathsf{s}_1 - \mathsf{s}_2 = \begin{cases} \mathsf{s} & \text{if } \mathsf{s}_1 = \mathsf{s}_2.\mathsf{s} \\ \bot & \text{otherwise} \end{cases}$$

Temporal Ordering for Session Environments

The $\Sigma \sqsubseteq \Sigma''$ notation means that the session environment Σ'' is at a later stage than the session environment Σ and is defined as thus.

$$\Sigma_1 \sqsubseteq \Sigma_2 \quad \text{iff}$$
$$\forall\, (\mathsf{cv}, \mathsf{ch}) \in \mathsf{dom}(\Sigma_2) : \quad \Sigma_1(\mathsf{cv}, \mathsf{ch}) = Udf \quad \vee \quad \Sigma_2(\mathsf{cv}, \mathsf{ch}) \preceq_{\mathsf{suf}} \Sigma_1(\mathsf{cv}, \mathsf{ch})$$

3.5.8 Static Typing of Expressions

We now state the the rules for static typing of the expressions together with an explanation of each rule. We have included the static typing rules of *Roles* in appendix C, for reference.

Axiom

$$\frac{}{\Gamma; \Sigma \vdash \mathsf{this} : \Gamma(\mathsf{this}); \Sigma}$$
$$\Gamma; \Sigma \vdash \mathsf{x} : \Gamma(\mathsf{x}); \Sigma$$

This rule asserts that the type of the `this` expression and the method argument (`x`) can be obtained from the standard environment Γ. In other words, `this` and `x` can be typed so long as their type is defined in Γ.

Null

$$\mathsf{Prog} \vdash \mathsf{C} \diamond_{\mathsf{c}}$$
$$\mathsf{Prog} \vdash \mathsf{CV} \diamond_{\mathsf{cv}}$$
$$\frac{}{\Gamma; \Sigma \vdash \mathsf{null} : \mathsf{C}; \Sigma}$$
$$\Gamma; \Sigma \vdash \mathsf{null} : \mathsf{CV}; \Sigma$$

The `null` expression have any class or conversation type so long as the class or conversation is defined as part of the program.

Subsumption

$$\mathsf{Prog} \vdash \mathsf{C} \leq \mathsf{C}'$$
$$\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{C}; \Sigma'$$
$$\frac{}{\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{C}'; \Sigma'}$$

This rule expresses the hierarchical relationship between a class and its superclass. If an expression `e` is of class type `C` and `C` itself is a subclass of class `C'`, we can deduce that `e` is also of type `C'`.

NewC

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\Gamma; \Sigma \vdash \mathsf{new\ C} : \mathsf{C}; \Sigma}$$

When creating a new instance of a class (`C`), if `C` is defined in the program, the new instance is of the class type `C`.

Seq

$$\frac{\Gamma; \Sigma \vdash \mathsf{e}_1 : \mathsf{t}_1; \Sigma'' \quad \Gamma; \Sigma'' \vdash \mathsf{e}_2 : \mathsf{t}_2; \Sigma'}{\Gamma; \Sigma \vdash \mathsf{e}_1; \mathsf{e}_2 : \mathsf{t}_2; \Sigma'}$$

When typing a sequence of expressions, provided that each expression in the sequence can be typed, the type of the sequence is the same as the type of the last expression in the chain. Furthermore, the initial session environment used to type each expression, is the final session environment returned by the previous expression in the chain.

Fld

$$\frac{\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{C}, \Sigma'}{\Gamma; \Sigma \vdash \mathsf{e.f} : \mathsf{F}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}); \Sigma'}$$

A field access expression can be typed if the receiver has a class type and the desired field is defined within the class at source level. The type of this expression is then type of the required field.

FldAssign

$$\frac{\Gamma; \Sigma \vdash e : C; \Sigma'' \quad \Gamma; \Sigma'' \vdash e' : F(Prog, C, f); \Sigma'}{\Gamma; \Sigma \vdash e.f = e' : F(Prog, C, f); \Sigma'}$$

When assigning to a field of a class, the receiver of the field must have a class type, the filed must be defined within that class and the type of the expression on the right hand side must be the same as the type of the field.

Method

$$\frac{\Gamma; \Sigma \vdash e : C; \Sigma_0 \quad \Gamma; \Sigma_{i-1} \vdash e_i : t_i; \Sigma_i \qquad i \in \{1...n\} \quad mType(Prog, C, m) = t_1...t_n \rightarrow t, \Sigma' \quad \Sigma' \preceq_{pre} \Sigma_n}{\Gamma; \Sigma \vdash e.m(e_1...e_n) : t; \Sigma_n - \Sigma'}$$

This rule is very similar to the standard typing rule for method calls. First the receiver of the method call is checked to ensure it is a class type and that the method is contained within its definition. Next we consider the types of arguments to determine if they agree with the type associated with them at the source level. Finally, we need to check if the session environment defined as a precondition in the method signature does not clash with the initial session environment. The definition of prefix session ordering is given in section 3.5.7. The assertion $\Sigma' \preceq_{pre} \Sigma_n$ states that any behaviour required by the method as the precondition (Σ') must also be permitted by the initial session environment (Σ_n). The final session environment is given by $\Sigma' - \Sigma_n$ since the session type associated with each entry in Σ' is consumed in the method body and is equivalent to ϵ upon exit from the method. Therefore, the final session environment is obtained by subtracting Σ' from Σ'_n where session subtraction is defined in section 3.5.7.

NewCV

$$\frac{\mathsf{Prog} \vdash \mathsf{CV}\diamond_{\mathsf{cv}}}{\Gamma; \Sigma \vdash \mathsf{new\ CV}() : \mathsf{CV}; \Sigma}$$

This rule asserts that as long as the conversation being instantiated is defined within the program, the `new CV()` expression has the type `CV`.

Spawn

$$\frac{\Gamma; \emptyset \vdash \mathsf{e} : \mathsf{t}; \emptyset}{\Gamma; \Sigma \vdash \mathsf{Spawn}\{\mathsf{e}\} : \mathsf{t}; \Sigma}$$

When spawning a new thread to execute an expression `e`, the `spawn` expression has the same type as the expression `e`.

Join

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \\ \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) \qquad \forall \mathsf{ch}\ \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r})\ :\ \Sigma_0(\mathsf{cv}, \mathsf{ch}) = Udf \\ \forall \mathsf{ch_i} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r})\ :\ \Sigma_\mathsf{i} = \Sigma_{\mathsf{i}-1}[(\mathsf{cv}, \mathsf{ch_i}) \mapsto \mathsf{session}(\mathsf{Prog}, \mathsf{CV}, \mathsf{ch_i})] \\ \mathsf{n}\ =\ \sharp\mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r})\end{array}}{\Gamma; \Sigma_0 \vdash \mathsf{cv.r.join}() : \mathsf{CV}; \Sigma_\mathsf{n}}$$

A process can only join a role in a conversation if it has not already joined it. In other words, the initial session environment must have no entries for any of the channels over which members of the role can communicate. This is expressed through the premise in the second line. Furthermore, when a new participant joins a role, we have to update the session environment by adding entries for the channels the new member can use as part of the role in question. Therefore, we obtain the final session environment Σ_n, by adding these new entries.

Leave

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \\ \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) \qquad \forall \mathsf{ch} \ \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) \ : \ \Sigma_0(\mathsf{cv}, \mathsf{ch}) \neq Udf \\ \forall \mathsf{ch_i} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) \ : \ \Sigma_\mathsf{i} = \Sigma_{\mathsf{i}-1}[(\mathsf{cv}, \mathsf{ch}) \mapsto \epsilon] \\ \mathsf{n} \ = \ \sharp\mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r})\end{array}}{\Gamma; \Sigma_0 \vdash \mathsf{cv.r.leave}() : \mathsf{CV}; \Sigma_\mathsf{n}}$$

A process can only leave a role in a conversation if it is already a member of the role. That is to say, the initial session environment must include an entry for each of the channels of the conversation over which members of the role communicate. This has been reflected in the premise at the end of the second line. Moreover, when a participant leaves a role, we have to update the session environment by removing entries for the channels associated with the role. Therefore we obtain the final session environment Σ_n by removing these entries.

Start

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \\ \forall \mathsf{ch} \in \mathsf{CH}(\mathsf{Prog}, \mathsf{CV}) : \ \mathsf{session}(\mathsf{Prog}, \mathsf{CV}, \mathsf{ch}) = \mathsf{s} \quad \rightarrow \quad \Sigma(\mathsf{cv}, \mathsf{ch}) = \mathsf{s}\end{array}}{\Gamma; \Sigma \vdash \mathsf{cv.start}() : \mathsf{CV}; \Sigma}$$

A conversation can only start if it has not already been initialised at an earlier point in time. This can be obtained by asserting that the session type prescribed to any of the channels in the initial session environment must be equal to its session defined at source level. If the session type of each channel is intact, one can deduce that as far as the communication actions are concerned, no participant has yet communicated any messages. In other words, even if the conversation has already started at some point in the past, the members of the various roles in the conversation have only carried out local computations and no communication has taken place yet. Undoubtedly, this is not strong enough to guarantee that the the conversation in question has not started, since intact session types only indicate absence of communication. However, this is the strongest we can type check a `start` expression statically. In section 3.6 we conjecture that our type system is sound so long as the `start` instruction is not called on an ongoing

conversation.

Send

$$\frac{\Gamma(\mathsf{cv}) = \mathsf{CV} \qquad \Gamma(\mathsf{v}) = \mathsf{C} \quad \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog}, \mathsf{CV}) \quad \Sigma(\mathsf{cv}, \mathsf{ch}) = !\mathsf{C}.\mathsf{s}}{\Gamma; \Sigma \vdash \mathsf{cv}.\mathsf{ch}.\mathsf{send}(\mathsf{v}) : \mathsf{C};\ \Sigma[(\mathsf{cv}, \mathsf{ch}) \mapsto \mathsf{s}]}$$

For a `send` expression to type check correctly, the session associated with the particular channel in the initial session environment must allow the sending of a value. This has been reflected in the premise on the third line. The final session environment is obtained by modifying the session prescribed to the channel in the initial session environment so that it does not accommodate the particular send action anymore.

Receive

$$\frac{\Gamma(\mathsf{cv}) = \mathsf{CV} \quad \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog}, \mathsf{CV}) \quad \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma' \quad \Gamma(\mathsf{x}) = \mathsf{C} \qquad \Sigma(\mathsf{cv}, \mathsf{ch}) = \Sigma'(\mathsf{cv}, \mathsf{ch}) = ?\mathsf{C}.\mathsf{s}}{\Gamma; \Sigma \vdash (\mathsf{x} = \mathsf{cv}.\mathsf{ch}.\mathsf{receive}()\ \mathsf{in}\ \mathsf{e}) : \mathsf{t};\ \Sigma'[(\mathsf{cv}, \mathsf{ch}) \mapsto \mathsf{s}]}$$

This is analogous to the typing rule for the `send` instruction. First the initial environment is checked to ensure the session type assigned to the send channel admits a receive action. The final session environment is then derived by updating the session type appointed to the channel and dropping the particular receive action. Since the `receive` expression entails an implicit loop, we can guarantee that *all* messages from the members of the opposite role have been acquired and it is safe to advance the session type.

3.5.9 Runtime Typing Judgement

In section 3.4, we introduced the operational semantics of the *Roles* language. We presented the rewriting rules of *Roles* as *small step* reductions. However, since the static type system put forward in the previous section applies typing judgements to source-level expressions, we need to perform additional typing checks on *intermediate* expressions derived at runtime from applying rewriting rules to source-level expressions. In this section we will present these supplementary typing rules forming the runtime type system.

Typing an expression at runtime is a judgement of the form:

$$\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{t}; \Sigma'$$

where the (e, π) pair refers to the currently active thread. We override the notion of Σ to denote the *runtime* session environment for the current thread (the one with π as the identifier) which is defined as follows where S has the usual meaning.

$$\Sigma ::= (\mathsf{ConvAddr} \times \mathsf{ChId}) \rightarrow \mathsf{S}$$

Note that the definitions given in the previous section for static session environments such as session ordering and session subtraction (cf. section 3.5.7) can be overridden to apply to runtime session environments by replacing the source level conversation instance (`cv`) with runtime conversation address κ.

3.5.10 Runtime Typing of Processes

In this section we present the runtime typing rules of the processes. We have included the runtime typing rules of *Roles* in appendix D, for reference. We skip the explanation of rules, since they are very similar in principle to the static typing rules specified in section 3.5.8.

Null

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}} \quad \mathsf{Prog} \vdash \mathsf{CV}\diamond_{\mathsf{cv}}}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{null}, \pi) : \mathsf{C}; \Sigma \quad \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{null}, \pi) : \mathsf{CV}; \Sigma}$$

EOM

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{EOM}, \pi) : \mathsf{C}; \Sigma}$$

ObjAddr

$$\frac{\chi(\iota) \downarrow_1 = \mathsf{C}}{\chi; \Sigma \vdash_{\mathsf{r}} (\iota, \pi) : \mathsf{C}; \Sigma}$$

ConvAddr

$$\frac{\chi(\kappa) \downarrow_1 = \mathsf{CV}}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa, \pi) : \mathsf{CV}; \Sigma}$$

Subsumption

$$\frac{\mathsf{Prog} \vdash \mathsf{C} \leq \mathsf{C}' \quad \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{C}; \Sigma'}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{C}'; \Sigma'}$$

NewC

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{new}\ \mathsf{C}, \pi) : \mathsf{C}; \Sigma}$$

Seq

$$\frac{\chi; \Sigma \vdash_r (e_1, \pi) : t_1; \Sigma'' \quad \chi; \Sigma'' \vdash_r (e_2, \pi) : t_2; \Sigma'}{\chi; \Sigma \vdash_r (e_1; e_2, \pi) : t_2; \Sigma'}$$

Fld

$$\frac{\chi; \Sigma \vdash_r (e, \pi) : C; \Sigma}{\chi; \Sigma \vdash_r (e.f, \pi) : F(Prog, C, f); \Sigma}$$

FldAssign

$$\frac{\chi; \Sigma \vdash_r (e, \pi) : C; \Sigma \quad \chi; \Sigma \vdash_r (e', \pi) : F(Prog, C, f); \Sigma'}{\chi; \Sigma \vdash_r (e.f := e',\ \pi) : F(Prog, C, f); \Sigma'}$$

Method

$$\frac{\begin{array}{l}\chi; \Sigma_0 \vdash_r (e, \pi) : C; \Sigma_0 \\ mType(Prog, C, m) = t_1 ... t_n \to t, \Sigma' \\ \chi; \Sigma_{i-1} \vdash_r (e_i, \pi) : t'_i; \Sigma_i \qquad Prog \vdash t'_i \leq t_i \qquad \text{for } i \in \{1...n\} \\ \Sigma' \preceq_{pre} \Sigma_n\end{array}}{\chi; \Sigma_0 \vdash_r (e.m(e_1 ... e_n), \pi) : t;\ \Sigma_n - \Sigma'}$$

NewCV

$$\frac{Prog \vdash CV \diamond_{cv}}{\chi; \Sigma \vdash_r (\text{new } CV(), \pi) : CV; \Sigma}$$

Spawn

$$\frac{\chi; \emptyset \vdash_{\mathsf{r}} (\mathsf{e}, \pi') : \mathsf{t}; \emptyset \qquad \text{for some } \pi' \notin \chi}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{Spawn}\{\mathsf{e}\}, \pi) : \mathsf{t}; \Sigma}$$

Join

$$\frac{\begin{array}{l}\chi(\kappa) \downarrow_1 = \mathsf{CV} \\ \pi \notin \chi(\kappa) \downarrow_2 (\mathsf{r}) \qquad \chi(\kappa) \downarrow_3 = \epsilon \\ \forall\, \mathsf{i}_{\{1 \le \mathsf{i} \le \sharp \mathsf{role\_channels(Prog,CV,r)} \,=\, \mathsf{n}\}} : \; \Sigma_{\mathsf{i}} = \Sigma_{\mathsf{i}-1}[(\kappa, \mathsf{ch_i}) \mapsto \mathsf{session}(\mathsf{Prog}, \mathsf{CV}, \mathsf{ch_i})]\end{array}}{\chi; \Sigma_0 \vdash_{\mathsf{r}} (\kappa.\mathsf{r.join}(), \pi) : \mathsf{CV}; \Sigma_{\mathsf{n}}}$$

Leave

$$\frac{\begin{array}{l}\chi(\kappa) \downarrow_1 = \mathsf{CV} \qquad \pi \in \chi(\kappa) \downarrow_2 (\mathsf{r}) \\ \forall\, \mathsf{i}_{\{1 \le \mathsf{i} \le \sharp \mathsf{role\_channels(Prog,CV,r)} \,=\, \mathsf{n}\}} : \; \Sigma_{\mathsf{i}} = \Sigma_{\mathsf{i}-1}[(\kappa, \mathsf{ch_i}) \mapsto \epsilon]\end{array}}{\chi; \Sigma_0 \vdash_{\mathsf{r}} (\kappa.\mathsf{r.leave}(), \pi) : \mathsf{CV}; \Sigma_{\mathsf{n}}}$$

Start

$$\frac{\chi(\kappa) \downarrow_1 = \mathsf{CV}}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa.\mathsf{start}(), \pi) : \mathsf{CV}; \Sigma}$$

Send

$$\frac{\begin{array}{ll}\chi(\kappa) \downarrow_1 = \mathsf{CV} & \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog}, \mathsf{CV}) \\ \Sigma(\kappa, \mathsf{ch}) = !\mathsf{C.s} & \chi; \emptyset \vdash_{\mathsf{r}} \mathsf{v} : \mathsf{C}'; \emptyset \\ \mathsf{Prog} \vdash \mathsf{C}' \le \mathsf{C} & \Sigma' = \Sigma[(\kappa, \mathsf{ch}) \mapsto \mathsf{s}]\end{array}}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa.\mathsf{ch.send}(\mathsf{v}), \pi) : \mathsf{C}; \Sigma'}$$

Receive

$$\frac{\chi(\kappa)\downarrow_1 = \mathsf{CV} \qquad \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog},\mathsf{CV}) \quad \chi;\Sigma \vdash (\mathsf{e},\pi) : \mathsf{t};\Sigma' \quad \Sigma(\kappa,\mathsf{ch}) = \Sigma'(\mathsf{cv},\mathsf{ch}) = ?\mathsf{C.s}}{\chi;\Sigma \vdash_{\mathsf{r}} (\ (\mathsf{let\ x} = \kappa.\mathsf{ch.receive()\ in\ e})\ ,\pi) : \mathsf{t};\Sigma'[(\mathsf{cv},\mathsf{ch}) \mapsto \epsilon]}$$

3.6 Soundness of *Roles* Type System

In order to show the type safety of the *Roles* language, we first need to define several notions such as well-formedness and agreement. In this section, we define these foundation concepts and then discuss type safety in the context of *Roles*.

3.6.1 Well-formedness

In this section we define the notion of well-formedness for programs and heaps and consequently for classes and conversations. These definitions have been inspired by the work of Drossopoulou in [20].

wfProg

$$\frac{\forall \mathsf{C} : \mathsf{Prog}(\mathsf{C}) \neq Udf \rightarrow \mathsf{Prog} \vdash \mathsf{C}\diamond \quad \forall \mathsf{CV} : \mathsf{Prog}(\mathsf{CV}) \neq Udf \rightarrow \mathsf{Prog} \vdash \mathsf{CV}\diamond}{\vdash \mathsf{Prog}\diamond}$$

wfConv

$$\frac{\forall \mathsf{ch}\ :\ \mathsf{CH}(\mathsf{Prog},\mathsf{CV},\mathsf{ch}) = (\mathsf{r}_1,\mathsf{r}_2,\mathsf{s}) \quad \longrightarrow \quad \mathsf{r}_1,\mathsf{r}_2 \in \mathsf{RS}(\mathsf{Prog},\mathsf{CV}) \ \wedge\ \mathsf{Prog} \vdash \mathsf{s}\diamond_{\mathsf{s}}}{\mathsf{Prog} \vdash \mathsf{CV}\diamond}$$

wfClass

$$\frac{\begin{array}{l}\mathsf{Prog} \vdash \diamond_{\mathsf{a}} \\ \mathsf{Prog}(\mathsf{C}) \downarrow_1 = \mathsf{C}' \ \text{ and } \ (\mathsf{C}' = \mathsf{Object} \ \text{ or } \ \mathsf{Prog}(\mathsf{C}') \neq Udf) \\ \forall \mathsf{f} \ : \ \mathsf{FD}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}) = \mathsf{t} \ \rightarrow \ \mathsf{Prog} \vdash \mathsf{t} \diamond_{\mathsf{t}} \ \text{ and } \ \mathsf{F}(\mathsf{Prog}, \mathsf{C}', \mathsf{f}) = Udf \\ \forall \mathsf{m} \ : \ \mathsf{M}(\mathsf{Prog}, \mathsf{C}, \mathsf{m}) = \mathsf{t} \ \mathsf{m}(\overline{\mathsf{t}\ \mathsf{x}}) \ \text{requires} \ \Sigma \ \{\mathsf{e}\} \ \longrightarrow \\ \qquad \mathsf{Prog} \vdash \mathsf{t} \diamond_{\mathsf{t}} \\ \qquad \mathsf{Prog} \vdash \mathsf{t}_{\mathsf{i}} \diamond_{\mathsf{t}} \qquad \text{for each } \mathsf{t}_{\mathsf{i}} \ \in \ \bar{\mathsf{t}} \\ \qquad \mathsf{Prog} \vdash \Sigma \diamond \\ \qquad \mathsf{Prog}, \ (\overline{\mathsf{t}\ \mathsf{x}}, \ \mathsf{C}\ \mathsf{this}), \Sigma \vdash \mathsf{e} : \mathsf{t}', \emptyset \ \longrightarrow \ \mathsf{t}' \leq \mathsf{t} \\ \qquad \mathsf{M}(\mathsf{Prog}, \mathsf{C}', \mathsf{m}) = Udf \ \text{ or } \ \mathsf{M}(\mathsf{Prog}, \mathsf{C}', \mathsf{m}) = \mathsf{t} \ \mathsf{m}(\overline{\mathsf{t}\ \mathsf{x}}) \ \text{requires} \ \Sigma \ \{\mathsf{e}'\}\end{array}}{\mathsf{Prog} \vdash \mathsf{C} \diamond}$$

wfSE

$$\frac{\forall \mathsf{cv}, \mathsf{ch} : \Sigma(\mathsf{cv}, \mathsf{ch}) = \mathsf{s} \ \rightarrow \ \mathsf{Prog} \vdash \ \mathsf{s} \diamond_{\mathsf{s}}}{\mathsf{Prog} \vdash \Sigma \diamond}$$

wfHeap

$$\frac{\begin{array}{l}\forall \ \iota \ : \ \chi(\iota) \downarrow_1 = \mathsf{C} \ \longrightarrow \mathsf{Prog}, \chi \vdash \iota \triangleleft \mathsf{C} \\ \forall \ \kappa \ : \ \chi(\kappa) \downarrow_1 = \mathsf{CV} \ \longrightarrow \mathsf{Prog}, \chi \vdash \kappa \triangleleft \mathsf{CV}\end{array}}{\mathsf{Prog} \vdash \chi \diamond}$$

Where the notion of agreement $\mathsf{Prog}, \chi \vdash \mathsf{v} \triangleleft \mathsf{t}$ is as defined in the section to follow.

3.6.2 Agreement

The judgement $\mathsf{Prog}, \chi \vdash \mathsf{v} \triangleleft \mathsf{t}$ expresses that the value v from heap χ *agrees* with the type t as defined in program Prog and is defined as follows.

$$\frac{\mathsf{Prog} \vdash \mathsf{t} \diamond_{\mathsf{c}} \quad \mathsf{Prog} \vdash \mathsf{t}' \diamond_{\mathsf{cv}}}{\begin{array}{l}\mathsf{Prog}, \chi \vdash \mathsf{null} \triangleleft \mathsf{t} \\ \mathsf{Prog}, \chi \vdash \mathsf{null} \triangleleft \mathsf{t}' \\ \mathsf{Prog}, \chi \vdash \mathsf{EOM} \triangleleft \mathsf{t}\end{array}}$$

The type of the null expression in the heap agrees with any class or conversation that is defined as part of the program. Similarly, the type of the EOM expression in the heap agrees with any class defined in the program. Note that we do not give the mirror expression stating the agreement between the type of the EOM expression and any constituent conversation of the program since EOM is a special value communicated over channels only and the values exchanged over channels can only have a *class* type.

$$\frac{\begin{array}{l}\chi(\iota) \downarrow_1 = \mathsf{C} \\ \mathsf{Prog}(\mathsf{C}) \neq Udf \\ \mathsf{F}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}) = \mathsf{C}' \longrightarrow \chi(\iota) \downarrow_2 (\mathsf{f}) = \mathsf{null} \ \text{ or } \ \chi(\chi(\iota) \downarrow_2 (\mathsf{f})) \downarrow_1 = \mathsf{C}' \\ \mathsf{F}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}) = \mathsf{CV} \longrightarrow \chi(\iota) \downarrow_2 (\mathsf{f}) = \mathsf{null} \ \text{ or } \ \chi(\chi(\iota) \downarrow_2 (\mathsf{f})) \downarrow_1 = \mathsf{CV}\end{array}}{\mathsf{Prog}, \chi \vdash \iota \triangleleft \mathsf{C}}$$

The type of an object address ι agrees with a class type when it satisfies the following properties:

- The first entry in its tuple representation must be a class defined in the program.
- For any of its fields f, if the type given to f at the source level is a class type C', the value recorded for f in the heap must be either null or an object address where the type of this address agrees with C'.
- For any of its fields f, if the type given to f at the source level is a conversation type CV, the value recorded for f in the heap must be either null or a conversation address where the type of this address agrees with CV.

$\chi(\kappa) \downarrow_1 = \mathsf{CV}$
$\forall\ \pi \in \chi(\kappa) \downarrow_2 (\mathsf{r}) :\ \forall \mathsf{ch} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) :$
$\forall\ \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) :\ \forall\ \pi \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) :$
$\quad \forall\ \mathsf{ch} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) :\ \chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k}) \ \ \wedge \ \ \mathsf{i}, \mathsf{j}, \mathsf{k} \in \mathsf{int}$
$\forall\ \mathsf{ch} \in \mathsf{chanIDs}(\mathsf{Prog}, \mathsf{CV}) :\ \forall\ \mathsf{i} \in \mathsf{int} :$
$\quad \forall\ \mathsf{v} \in \chi(\kappa) \downarrow_3 (\mathsf{ch}, \mathsf{i}) :\ \mathsf{Prog}, \chi \vdash \mathsf{v} \triangleleft \mathsf{out\_T}(\mathsf{session}(\mathsf{Prog}, \mathsf{CV}, \mathsf{ch}),\ \mathsf{i})$

$\mathsf{Prog}, \chi \vdash \kappa \triangleleft \mathsf{CV}$

The type of a conversation address (κ) agrees with a conversation type if it satisfies the following conditions.

- The first entry in its tuple representation must be recorded as a conversation defined in the program.
- The indexing information recorded in the heap for each of the participants of its constituent roles must be a triple of integers.
- The type of the values communicated over its channels must agree with the corresponding types specified in the session type of the channel.

In order to show that the type of expressions is preserved after each execuion step, we also need to define the notion of agreement between the environments and the heap. The following asserts the agreement between session environment Σ and the currently active process π and heap χ written as $\mathsf{Prog}; \Sigma \vdash (\chi, \pi) \diamond$.

$$
\begin{aligned}
\forall (\mathsf{cv}, \mathsf{ch}) :\ \Sigma(\mathsf{cv}, \mathsf{ch}) = & !\mathsf{C.s} \rightarrow \\
& \chi(\pi, \mathsf{cv}, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k}) \quad \wedge \quad \chi(\mathsf{cv}) \downarrow_3 (\mathsf{ch}, \mathsf{i}) = \overline{\mathsf{v}} \\
& \wedge\ \forall\ \mathsf{v} \in \overline{\mathsf{v}} :\ \ \mathsf{Prog}, \chi \vdash \mathsf{v} \triangleleft \mathsf{C} \\
\Sigma(\mathsf{cv}, \mathsf{ch}) = & ?\mathsf{C.s} \rightarrow \\
& \chi(\pi, \mathsf{cv}, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k}) \quad \wedge \quad \chi(\mathsf{cv}) \downarrow_3 (\mathsf{dual}(\mathsf{ch}), \mathsf{rStage}) = \overline{\mathsf{v}} \\
& \wedge\ \forall\ \mathsf{v} \in \overline{\mathsf{v}} :\ \ \mathsf{Prog}, \chi \vdash \mathsf{v} \triangleleft \mathsf{C} \\
& \mathsf{where}
\end{aligned}
$$

$$
\mathrm{rStage} = \begin{cases} \mathsf{j} & \text{if k } < \ \sharp\text{senders(Prog, } \chi\text{, cv, ch) - 1} \\ \mathsf{j{+}1} & \text{if k } = \ \sharp\text{senders(Prog, } \chi\text{, cv, ch) - 1} \end{cases}
$$

$\mathsf{Prog}, \Sigma \vdash (\chi, \pi) \diamond$

3.6.3 Properties of the Heap

Throughout designing of the *Roles* language and while deciding between possible ways of representing the channels of a conversation in the heap, we observed the following properties about the heap. Note that in some of the following properties we have made use of some of the auxiliary functions defined in section 3.3.

Property 1 (Channel Duals)

This statement asserts that for any conversation instance, if a channel is defined for that instance in the heap, its dual is also defined. In other words, upon initiation of the conversation, for each channel defined at the source level, both endpoints (the channel and its dual) are defined in the heap.

$$\forall\, \chi, \kappa, \mathsf{ch} : \mathsf{ch} \in \mathsf{dom}(\chi(\kappa)\downarrow_3) \;\longrightarrow\; \mathsf{dual}(\mathsf{ch}) \in \mathsf{dom}(\chi(\kappa)\downarrow_3)$$

Property 2 (Message Cardinality)

This conjecture suggests that for any channel in the heap, the number of messages on its notice board at any round of communication is at most equal to the number of participants sending on that channel.

$$\begin{aligned}\forall\, \kappa, \mathsf{ch}, \mathsf{i} :\; & \chi(\kappa)\downarrow_3 (\mathsf{ch}, \mathsf{i}) = \mathsf{v}^* \\ & \longrightarrow \sharp\mathsf{v}^* \;\leq\; \sharp\mathsf{senders}(\mathsf{Prog}, \chi, \kappa,\; \mathsf{ch})\end{aligned}$$

Property 3 (Channel Index Ordering)

The following statement asserts the relationship between the sending/receiving stages of the participants of each role and their dual. Assume in the heap χ, for the conversation instance κ, participants of role $\mathtt{r_1}$ communicate over channel $\mathtt{ch}$ with participants of role $\mathtt{r_2}$ and π_1 and π_2 are participants of roles $\mathtt{r_1}$ and $\mathtt{r_2}$ respectively. If the $(\pi_1, \kappa, \mathtt{ch})$ and $(\pi_2, \kappa, \mathtt{d(ch)})$ entries are mapped to $(\mathtt{sStage_1}, \mathtt{rStage_1}, \mathtt{rInd_1})$ and $(\mathtt{sStage_2}, \mathtt{rStage_2}, \mathtt{rInd_2})$ tuples in the heap respectively, then we must have $\mathtt{rStage_1} \leq \mathtt{sStage_2}$, since as a participant of role $\mathtt{r_1}$, π_1 is expected to receive values sent over channel $\mathtt{ch}$ from *all* participants of role $\mathtt{r_2}$ including π_2. Therefore, π_1's receiving index should be no more than the sending index of *any* of the participants in role $\mathtt{r_2}$. If π_1's receiving index is more than π_2's sending index, π_1 has indeed

missed the communicated value from π_2 and thus has not received values from *all* participants of $\mathtt{r_2}$.

Furthermore, we must have $\mathtt{rInd_1} \leq \sharp\mathtt{senders}(\mathtt{Prog}, \chi, \kappa, \mathtt{d}(\mathtt{ch}))$, since receiving index refers to the number of messages π_1 has received so far as part of the current receiving stage ($\mathtt{rStage_1}$) and it can be no more than the number of participants in role $\mathtt{r_2}$ as each message is received *exactly once*.

$$\begin{aligned}\forall\, \chi, \kappa, \pi_1, \pi_2, \mathsf{ch} \;:\quad & \chi(\pi_1, \kappa, \mathsf{ch}) = (\mathsf{i}_1, \mathsf{j}_1, \mathsf{k}_1) \;\wedge\; \chi(\pi_2, \kappa, \mathsf{dual}(\mathsf{ch})) = (\mathsf{i}_2, \mathsf{j}_2, \mathsf{k}_2) \\ & \qquad\qquad \longrightarrow \\ & \mathsf{j}_1 \leq \mathsf{i}_2 \;\wedge\; \mathsf{k}_1 \leq \sharp\mathsf{senders}(\mathsf{Prog}, \chi, \kappa, \mathsf{dual}(\mathsf{ch})) \\ & \mathsf{j}_2 \leq \mathsf{i}_1 \;\wedge\; \mathsf{k}_2 \leq \sharp\mathsf{senders}(\mathsf{Prog}, \chi, \kappa, \mathsf{ch})\end{aligned}$$

Property 4 (Channels in the Heap)

Any channel that is defined as part of a conversation instance in the heap must also have been defined in the source code of the conversation.

$$\begin{aligned}\mathsf{I}_3(\chi) = \forall\, \kappa, \mathsf{ch} \;:\; & \chi(\kappa)\downarrow_1 = \mathsf{CV} \;\wedge\; \chi(\kappa)\downarrow_3 \neq \mathsf{Udf} \\ & \longrightarrow \mathsf{ch} \in \mathsf{ChanIds}(\mathsf{Prog}, \mathsf{CV})]\end{aligned}$$

Property 5 (Channels at Source Code)

This is the reverse of the previous conjecture. For any channel contained within a conversation's definition in the source code, every instance of that conversation in the heap etiher contains a definition for that channel, or has not started yet.

$$\begin{aligned}\mathsf{I}_4(\chi) = \forall\, \kappa, \mathsf{ch} \;:\; & \chi(\kappa)\downarrow_1 = \mathsf{CV} \;\wedge\; \mathsf{ch} \in \mathsf{ChanIds}(\mathsf{Prog}, \mathsf{CV}) \\ & \longrightarrow \chi(\kappa)\downarrow_3 = \epsilon \;\vee\; \chi(\kappa)\downarrow_3 (\mathsf{ch}) \neq \mathsf{Udf}\end{aligned}$$

Throughout execution, a heap is modified as specified in the reduction rules in section 3.4. We call a heap *synchronised* if it satisfies certain properties at all points during execution. Below we state these properties which govern the relationship between the channel indexing information of different threads.

Property 6 (Synchronised Heap)

If the session type of a channel for a role consists of a chain of send actions (possibly singular) followed by a receive action, all participants of that role must finish sending as part of the *last* send action in the chain before performing the following receive action.

$$\begin{aligned}\forall\, \kappa, \pi_1, \pi_2, \mathsf{r}, \mathsf{ch} \;:\;\; & \pi_1, \pi_2 \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) \\ & \wedge\, \chi(\pi_1, \kappa, \mathsf{ch}) = (\mathsf{i}_1, \mathsf{j}_1, \mathsf{k}_1) \;\wedge\; \chi(\pi_2, \kappa, \mathsf{ch}) = (\mathsf{i}_2, \mathsf{j}_2, \mathsf{k}_2) \\ & \wedge\, \mathsf{i}_1 \neq \mathsf{i}_2 \\ & \longrightarrow \mathsf{j}_1 = \mathsf{j}_2 \;\wedge\; \mathsf{k}_1 = \mathsf{k}_2 = 0\end{aligned}$$

Properties 7 and 8 (Synchronised Heap)

If the session type of a channel for a role consists of a chain of receive actions (possibly singular) followed by a send action, all participants of that role must finish receiving all messages as part of each receive action in the chain before performing the following send action.

$$\begin{aligned}\forall\, \kappa, \pi_1, \pi_2, \mathsf{r}, \mathsf{ch} \;:\;\; & \pi_1, \pi_2 \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) \\ & \wedge\, \chi(\pi_1, \kappa, \mathsf{ch}) = (\mathsf{i}_1, \mathsf{j}_1, \mathsf{k}_1) \;\wedge\; \chi(\pi_2, \kappa, \mathsf{ch}) = (\mathsf{i}_2, \mathsf{j}_2, \mathsf{k}_2) \\ & \wedge\, \mathsf{j}_1 \neq \mathsf{j}_2 \\ & \longrightarrow \mathsf{i}_1 = \mathsf{i}_2\end{aligned}$$

$$\begin{aligned}\forall\, \kappa, \pi_1, \pi_2, \mathsf{r}, \mathsf{ch} \;:\;\; & \pi_1, \pi_2 \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) \\ & \wedge\, \chi(\pi_1, \kappa, \mathsf{ch}) = (\mathsf{i}_1, \mathsf{j}_1, \mathsf{k}_1) \;\wedge\; \chi(\pi_2, \kappa, \mathsf{ch}) = (\mathsf{i}_2, \mathsf{j}_2, \mathsf{k}_2) \\ & \wedge\, \mathsf{j}_1 = \mathsf{j}_2 \;\wedge\; \mathsf{k}_1 \neq \mathsf{k}_2 \\ & \longrightarrow \mathsf{i}_1 = \mathsf{i}_2\end{aligned}$$

3.6.4 Execution and Session Environments

The following expresses the relation between the initial session environment Σ and the final session environment Σ' when typing an expression e.

$$\Gamma, \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma' \;\;\rightarrow\;\; \Sigma' \sqsubseteq \Sigma$$

where the $\Sigma' \sqsubseteq \Sigma$ notion is defined in section 3.5.7.

3.6.5 Definition

$$\chi; \Sigma \vdash (\mathsf{e}, \pi) : \mathsf{t} || \Sigma' \;\;\textit{iff}\;\; \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{t}; \Sigma' \;\wedge\; \mathsf{prog}; \Sigma \vdash (\chi, \pi)\diamond$$

3.6.6 Subject Reduction Conjecture

The subject reduction theorem constitute a very important part of a type system since once it is proved, one can deduce the type safety of the system. In summary, the subject reduction theorem asserts that the type of expressions is invariant to the execution. In other words, if an expression can be typed, its type remains unchanged at any point during the execution.

We consider the reductions of well-typed expressions in our language and conjecture that the type of expressions is preserved after each execution step. An expression is well-typed if it can be typed by the rules stated in section 3.5.8.

- $\chi; \Sigma_i \vdash P_i : t_i || \Sigma'_i \qquad \text{for } i \in \{0...n\}$
  $\wedge \quad (P_0 | \overline{P}^{\,i=1...n}), \chi \quad \longrightarrow \quad (P'_0 \mid \overline{P}^{\,i=1...n} \mid \overline{P'}^{\,j=1...m}), \chi'$

  implies

  $\exists\, \Sigma''_0 : \ \chi'; \Sigma''_0 \vdash P_0 : t || \Sigma'_0 \ \wedge \ \Sigma_0 \sqsubseteq \Sigma''_0$
  $\wedge \quad \chi'; \Sigma_i \vdash P_i : t_i || \Sigma'_i \qquad \text{for } i \in \{1...n\}$
  $\wedge \quad \exists\, \Sigma''_j; t'_j : \ \chi'; \emptyset \vdash P'_j : t'_j || \Sigma''_j \qquad \text{for } j \in \{1...m\}$

- $\chi; \Sigma_i \vdash P_i : t_i || \Sigma'_i \qquad \text{for } i \in \{1...n\}$
  $\wedge \quad (\overline{P}^{\,i=1...n}), \chi \quad \longrightarrow^* \quad (\overline{P'}^{\,i=1...n} \mid \overline{P''}^{\,j=1...m}), \chi'$

  implies

  $\exists\, \Sigma''_i : \ \chi'; \Sigma''_i \vdash P_i : t_i || \Sigma'_i \ \wedge \ \Sigma_i \sqsubseteq \Sigma''_i \qquad \text{for } i \in \{1...n\}$
  $\wedge \quad \exists\, \Sigma'''_j; t'_j : \ \chi'; \emptyset \vdash P''_j : t'_j || \Sigma'''_j \qquad \text{for } j \in \{1...m\}$

The first part of the subject reduction conjecture can be explained as follows. Suppose there are $n+1$ processes currently present in the heap χ. Moreover, assume that the session environment associated with each of the processes agrees with the heap χ and that each of the processes can be typed given the heap χ and their respective session environments.

Finally, assume that given χ, one of these processes, namely process $\mathsf{P_0}$, can be rewritten to $\mathsf{P_0'}$ in a *single* reduction step, in doing so, the heap χ is modified, the new heap χ' is obtained and a group of m new processes ($\overline{\mathsf{P'}}$) are forked and join the system.

We can then deduce that the each of the processes present in the system (including those generated in the last reduction step) can also be typed given their respective session environments and the *new* heap χ', the type we obtain for each one of them is the same as the type we had before reduction, and that each of these session environments agrees with the new heap χ'.

The second part on the other hand, describes a more general case where *many* reduction steps are considered. In other words, it asserts that if the processes in the heap can be typed, if their session environments agree with the heap and if we can perform many reduction steps on these processes, then the resulting processes can also be typed, their type is unchanged and their session environments agree with the final heap.

Explanation of the $\Sigma_0 \sqsubseteq \Sigma_0''$ Statement

In what follows we give an explanation for the $\Sigma_0 \sqsubseteq \Sigma_0''$ assertion in the first part of the subject reduction conjecture through an example. Analogous assertions in the second part of the conjecture can be justified similarly.

Assume we have $\Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma'$. While executing e, we will evaluate it in small steps. In other words we have:

$$(\mathsf{e}, \pi), \chi \rightarrow \ (\mathsf{e'}, \pi), \chi' \rightarrow \ (\mathsf{e''}, \pi), \chi'' \rightarrow \ ... \rightarrow \ (\mathsf{v}, \pi), \chi_{\mathsf{n}}$$

which is often written as $(\mathsf{e}, \pi), \chi \rightarrow^* \ (\mathsf{v}, \pi), \chi_{\mathsf{n}}$. However, each step of execution may modify the session environment. Therefore, we have to ensure that at each intermediate step of execution the new session environment (in this case Σ'') reflects the changes brought upon the initial session environment (Σ) through execution. The instructions which modify the session environment are join, leave, send, receive and method call. When sending and receiving, the session of the desired channel in Σ is consumed and hence we have $\Sigma'' \preceq_{suf} \Sigma$. Leaving a role removes the said role's channel sessions from Σ and hence $\Sigma'' \preceq_{suf} \Sigma$. On the other hand, joining a role results in new entries being added to Σ, namely the sessions of the channels the said role communicates over.

In other words:

$$\exists\,(\mathsf{cv},\mathsf{ch}):\ \Sigma(\mathsf{cv},\mathsf{ch}) = Udf\ \ \wedge\ \ \Sigma(\mathsf{cv},\mathsf{ch}) \neq Udf.$$

Finally, method calls result in a session environment derived from subtracting the said methods *required* session environment from the initial session environment and thus we have $\Sigma'' \preceq_{suf} \Sigma$.
Therefore, we assert $\Sigma \sqsubseteq\ \Sigma''$ meaning that the secondary session environment Σ'' must be at a later stage than the initial session environment Σ.

3.6.7 Subderivation Conjecture

The subderivation conjecture asserts that if an expression `e` can be typed in an evaluation context, then the expression `e` can also be typed outside the evaluation context. However, the types given to the expression in the two different cases are not necessarily the same. This conjecture can be formalised as follows.

$$\chi; \Sigma_1 \vdash_{\mathsf{r}} (\mathsf{Ctxt}[\mathsf{e}], \pi) : \mathsf{t}; \Sigma_2$$

implies

$$\exists\ \Sigma_1', \Sigma_2':\ \Sigma_1' \preceq_{\mathsf{pre}} \Sigma_1\ \ \wedge\ \ \Sigma_2' \preceq_{\mathsf{pre}} \Sigma_2\ \ \wedge\ \ \chi; \Sigma_1' \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{t}'; \Sigma_2'$$

We now give an example to clarify the assertions of this conjecture. Assume that the two classes `C1` and `C2` are defined as in Figure 3.9. The expression `e = new C2()` is of type `C2` according to the typing rules in section 3.5.10. However in the `Ctxt1 = -.f1` context, `e` is of type `C1`, since `f1` is a field of class `C1` and `C2` inherits `C1`.

```
class C1 {
  String f1;
}

class C2 extends C1{
  int f2;
}
```

Figure 3.9: Example program in *Roles* consisting of two classes

3.7 Communication Safety

In this section we discuss another property of the *Roles* language. We first introduce definitions used in the description of the property and then we conjecture that the *Roles* language demonstrates the *progress* property. The progress property asserts that a well-typed expression (one that can be typed by the rules in section 3.5.10) never gets "stuck". In other words, a well-typed expression never gets into an undefined state where no further transitions are possible.

3.7.1 Nullpointer Failure

An expression e is a *nullpointer failure*, if it has one of the following forms.

- $\mathsf{Ctxt}[\mathsf{null.f}]$;
- $\mathsf{Ctxt}[\mathsf{null.m}(\bar{\mathsf{e}})]$;
- $\mathsf{Ctxt}[\mathsf{null.ch.send}(\mathsf{e})]$;
- $\mathsf{Ctxt}[\mathsf{null.ch.receive}()]$;
- $\mathsf{Ctxt}[\mathsf{null.r.join}()]$;
- $\mathsf{Ctxt}[\mathsf{null.r.leave}()]$;
- $\mathsf{Ctxt}[\mathsf{null.start}()]$;

3.7.2 Definition (Liveness)

A Process $\mathtt{P} = (\mathsf{e}, \pi)$ is called *alive* in heap χ written as alive($\mathtt{P}$, χ), if and only if:

- e is a value
- or e is a nullpointer failure
- or $\mathsf{e}, \chi \rightarrow \mathsf{e}', \chi'$
- or $\mathtt{e} = \mathtt{Ctxt}[\kappa.\mathtt{r.join}()]$ for some context `Ctxt[-]` and $\mathtt{hasStarted}(\chi, \kappa) = \mathtt{true}$.
- or $\mathtt{e} = \mathtt{Ctxt}[\kappa.\mathtt{start}()]$ for some context `Ctxt[-]` and $\mathtt{hasStarted}(\chi, \kappa) = \mathtt{true}$.
- or $\mathtt{e} = \mathtt{Ctxt}[\kappa.\mathtt{ch.send}(\mathtt{e}')]$ or $\mathtt{e} = \mathtt{Ctxt}[\kappa.\mathtt{ch.receive}()]$ for some context `Ctxt[-]` and $\mathtt{hasStarted}(\chi, \kappa) = \mathtt{false}$.

The fourth situation arises when expression e has been blocked while trying to join an ongoing conversation. The next item refers to the situation where e has been blocked while trying to start a conversation that has already started and the last item pertains to the case where e has been blocked while trying to send/receive a value over a channel in a conversation that has not yet started.

3.7.3 Progress Conjecture

The progress property asserts that well-typed processes never get stuck. In other words:

- $\chi; \Sigma_i \vdash P_i : t_i || \Sigma'_i \qquad \text{for } i \in \{0...n\}$
  $\wedge \quad (P_0 | \overline{P}^{\,i=1...n}), \chi \longrightarrow (P'_0 \mid \overline{P}^{\,i=1...n} \mid \overline{P'}^{\,j=1...m}), \chi'$

  implies

  $\mathsf{alive}(P'_0, \chi') \quad \wedge \quad \forall P'_j \in \overline{P'}^{\,j=1...m} : \quad \mathsf{alive}(P'_j, \chi')$

- $\chi; \Sigma_i \vdash P_i : t_i || \Sigma'_i \qquad \text{for } i \in \{1...n\}$
  $\wedge \quad (\overline{P}^{\,i=1...n}), \chi \quad \longrightarrow^* \quad (\overline{P'}^{\,i=1...n} \mid \overline{P''}^{\,j=1...m}), \chi'$

  implies

  $\forall P'_i \in \overline{P'}^{\,i=1...n} : \quad \mathsf{alive}(P'_i, \chi') \ \wedge$
  $\forall P''_j \in \overline{P'}^{\,j=1...m} : \quad \mathsf{alive}(P''_j, \chi')$

The first part of this conjecture asserts that if there are $\mathsf{n}+1$ *well-typed* processes currently present in the heap and one of them (P_0) performs a reduction and in doing so creates new processes, then the reduced process and the new processes are all alive.

The second part describes a similar scenario but for many reduction steps.

3.8 Design Decisions and Alternatives

When designing our language based on the *Roles* idea discussed above, there are certain decisions we had to make about the properties and functionality of our language. These decisions will strongly affect the expressiveness and conciseness of our notation and it is crucial to give them due consideration. Some of these decisions are explored below.

3.8.1 Roles Arity

The number of members in a particular role by definition ranges from one to many and we do not differentiate between roles with exactly a single member at all times and those with varying number of participants. Roles can accommodate one or many participants and this number can change while the communication lasts.

An alternative approach would have been to separate singleton roles from those with many participants. However, treating participants as individuals is a feature that is already available in existing systems and *Roles* is based on the idea of inter-group communication and defining the behaviour of a *group* of partcipants. Furthermore, *Roles* indeed accommodates roles with a single participant since there is no restriction on the arity of roles and any role is allowed to have a single participant. We believe this is integral to *Roles* and the multiplicity of the roles is a prominent feature of the language.

3.8.2 Empty Roles

Another issue to consider regarding the cardinality of a role is what to do when all participants of a particular role leave resulting in an *empty* role. We believe it is the programmer's responsibility to decide what to do in the event of such occurrences since we present a framework for interactions within which programs are written; we make no guarantees about the programs themselves.

An other approach would have been to forcibly end the conversation when any of the roles becomes empty. However, we have allowed more flexibility in *Roles* and roles with no participants can be detected due to lack of communication. When members of a role are expecting messages from an empty role, they will immediately receive an `EOM` message indicating there is no message available to them due to the absence of participants. The user can then decide whether he wishes to continue with an empty role, add more participants to the empty role or simply terminate the conversation.

3.8.3 Inter-Role Channels

We define a dedicated channel between each pair of communicating roles rather than allocating a channel to each role. Even though our approach leads to more channels, defining session types for channels is easier since we can express the behaviour of each role with respect to a single other role rather than having to *globally* define the overall behaviour of each role.

We believe it is much simpler to envisage the communication between constituent roles locally and with respect to each other. Moreover, since each channel is between two roles only, the recipients of messages are always aware of the identity of senders. On the other hand, if each role was assigned a dedicated channel, each message sent would have to identify the sender of the message by including additional information in the message about the role they belong to.

3.8.4 Messaging Subgroups and Individuals

The communication in the *Roles* language is broadcast by definition; that is to say, every message addressed at a particular role will be received by all members of that role. However, how does our language support for sending a message to a particular individual or a subgroup of members in a role? We delegate the responsibility of catering for subgroup messaging to the programmer through definition of necessary roles. In other words, if the programmer expects role $\mathtt{r}_1$ to communicate with a subgroup `x` of role $\mathtt{r}_2$ at some point in the future, he is expected to create a dedicated role for this group namely the `x` role and his expected behaviour upon initiation of the conversation. When role $\mathtt{r}_1$ wishes to talk to the said subgroup, he can address the members in the `x` role and hence the subgroup communication commences.

3.8.5 Underlying Paradigm

We adopt the object-oriented paradigm for the specification of the *Roles* language.

3.8.6 Synchronicity

In design of *Roles* we have adopted an *asynchronous* (non-blocking) approach since it is more flexible. When participants of a role send a message, they drop it in the recipients mail box and do not wait for the members of the opposite role to receive it.

3.8.7 Progress

In section 3.7 we made a conjecture that the *Roles* language has the progress property. However, as we discuss in section 5.2, further work needs to be done towards proving this property.

3.8.8 Role Participants

We have designed *Roles* so that the participants of each role are processes. Storing the identity of threads as members of roles is crucial since otherwise we cannot ensure safe communication. Suppose we reach a `send` instruction such as `cv.ch.send(v)`. If the identity of threads is not recorded as partcipants of roles, we cannot verify if this instruction is legal and if the currently active thread can access the channel `ch` of conversation `cv`. Depending on the control flow of the program, communication instructions can be reached at various locations and the only way to authorise these actions is by keeping track of the threads in each role and hence checking their channel access permissions. The expressions following the join statement stand for the behaviour of the new member and are executed upon joining.

An other possibility is to populate roles with objects alongside threads. This way the behaviour of members can be defined through methods of the object and hence reusing the code when the same behaviour is expected from several participants. Nevertheless, this can be simulated by simply calling the said method following the join instruction. Thus, stroing objects as well as threads suggests no real benefit but clutters the heap with unnecessary information. We believe recording thread identities as the participants of roles is the most concise and elegant way.

3.8.9 Spawning New Threads

We have included the `spawn` instruction in our syntax so that the user can fork new threads as required. Therefore, we expect the user to spawn a thread whenever a new participant is needed. Upon initiation of the conversation, threads representing participants of various roles start execution in parallel.

Alternatively, we could fork a new thread each time a join statement is reached. However, this way a thread can be a member of a single role only since each join statement would lead to spawning a new thread and hence one role per thread. Our approach allows for each thread to participate in multiple roles, since each time a `join` instruction is reached, it is the

currently active thread that joins the said role and joining multiple roles can be achieved by having multiple join statements.

3.8.10 Role Representation

In *Roles* the declaration of a role in a conversation at source level consists of the name of the role only. Initially we had designed *Roles* so that each role was given a class type to acquire uniformity across members of a role if necessary. However, at later revisions it was evident that associating each role with a class type did not present any benefit since the participants of roles are processes and can exhibit various behaviours. Therefore, we refined our notation accordingly.

3.8.11 Dynamic Leaving and Joining

One of the motivations of designing *Roles* was to accommodate dynamic joining and leaving of the participants. Our current system allows participants to dynamically leave a conversation at any point during execution. However, at this stage new members can only join a conversation if it has not already started. Once a conversation is initiated, joining any of its roles is no longer permitted. Nonetheless, as discussed in section 5.2, dynamic joining is an important feature of the *Roles* language and one can extend *Roles* to promote dynamic joining of new members.

3.8.12 Message Boxes

In *Roles* we have allocated two message boxes to each channel: one per participating role. When a message is sent to a role, it is dropped in the role's message box and participating members will retrieve their messages from this public "noticeboard". Participants maintain indexing information to determine where in the message box to look for the next message, since once a member has retrieved a particular message he does not delete it in case it has not been received by all participants. Therefore, indexing information is necessary to avoid duplicate messages.

Undoubtedly, another obvious solution is to allocate a message box per participant so that when a message is sent to a role it is delivered to the message box of each member. However, this approach has many shortcomings. For instance, whenever a member of a role performs a send action, he needs to deliver its message to each and every participant of the opposite role. In systems with large number of participants, this process can take a

large amount of time resulting in a slow system. Moreover, having seperate message boxes for each participant entails having multiple copies of the same data in the heap. We believe this is a very inefficient use of space since it clutters the heap with redundant information. This can be avoided if a suitable indexing technique is in place and the messages sent to a role is shared by all members through use of a common message box.

The design space guided many of the decisions made in the specification of the *Roles* language. In the next chapter we explore the outcomes of the design decisions taken throughout the specification of the *Roles* language and compare it against its contemporaries.

Conclusions

In this chapter we have introduced the concept of *Roles* and specified the foundations of the language. We first discuss the notions of a conversation and a role, and contrast them to the approaches discussed in chapter 2. Then we defined the basic syntax that makes up our language, as well as the operational semantics and type system of the new concepts we have introduced. We discussed these in depth, before conjecturing properties of the type system that we believe *Roles* benefits from. Finally, we debated the design decisions that we made during the development of *Roles*, and discussed what alternatives arose and why they were not chosen. In the next chapter we will look back on our work and evaluate it in detail.

Chapter 4

Evaluation

Previous chapters have included an in-depth summary of the theory surrounding and motivating our work on *Roles* as well as the specification of the language itself. We specified the syntax, the operational semantics and the type system of the *Roles* language in chapter 3. In the subsequent sections, we step through each of these in turn and assess them by comparing them against contemporary approaches discussed in chapter 2. We discuss the criteria for evaluation and the means by which we ensure these have been fulfilled.

4.1 *Roles* Syntax

In order to evaluate the syntax of *Roles*, we will assess it according to the following properties:

- A language's *expressiveness* is assessed on the complexity of the scenarios and changes it can describe. An expressive language can model complicated concepts without modification or extension.
- A language's *conciseness* is assessed on the size of the expressions required to represent concepts. A concise language can express its basic concepts with very little written syntax.
- A language's *readability* is assessed on the clarity with which it expresses meaning. Readable languages do not have excessive numbers of operators and can be both read and written quickly and clearly.

In order to compare the expressiveness, conciseness and readability of *Roles* with the existing approaches discussed in chapter 2, we rewrite com-

mon examples from the literature and those listed in chapter 2 also. Our intention is to produce a variety of scenarios that capture the various features of the *Roles* language, particularly those that it has in common with the other languages we have discussed in chapter 2. By doing this, we can compare the languages directly to evaluate whether *Roles* is more concise or expressive. We also wish to ensure that we at least do not *lose* any expressiveness - any example that is expressible in another session type language should also expressible in *Roles*.

We also present examples that exhibit the new features that are unique to *Roles*. One example of this is a scenario that demonstrates dynamic leaving of participants midway through a conversation; or an example to show broadcast communication, where a particular participant sends the same message to many recipients at once.

In the examples to follow, the parts of the code written in blue denote syntax that is exclusive to the language of *Roles* rather than generic object-oriented languages. On the other hand, code written in red indicates statements pertaining to session types or communication actions.

4.1.1 Auction Example

In section 2.5.4 we presented the auction communication system as well as its description in the syntax of dyadic session types. Later in section 2.6.4 we described this protocol in the notation of global session types. In what follows we will rewrite this in the syntax of the *Roles* language. We have assumed the existence of array constructs as well as auxiliary methods such as `getItems()`, `max()` and `getMyBid()` to simplify the example.

The auction protocol is defined in the `Auction` conversation written in *Roles* syntax as given in Figure 4.1. It consists of three roles, namely `Auctioneer, Audience` and `Bidder` as well as channels (`ab, aa`) between the auctioneer and the bidders and the auctioneer and the audience respectively.

Note that in each channel definition, the specified session type denotes the communication between the two parties from the *first* role's perspective. For instance, in the `ab` channel declaration the prescribed session denotes the behaviour of the `Auctioneer` role and the behaviour of the `Bidder` role is given by the *dual* of this session which is obtained by replacing every `!` with `?` and vice versa. Furthermore, a name given to a channel is the *endpoint* used by the *first* role and the other endpoint is referred to by calling the dual endpoint. For instance, in the `ab` channel declaration, the participant(s) of

the `Auctioneer` role use the `ab` endpoint to access this channel, whereas the participants of the `Bidder` role use `d(ab)` endpoint.

In this example we have assumed that there are two items on auction. Hence, the session type associated with the `ab` channel is given as:

```
!Item.?Bid.!Bid.!Item.?Bid.!Bid.
```

In other words, the session consists of *two* cycles of `!Item.?Bid.!Bid` where the auctioneer announces the item on auction (`!Item`), receives bids from all bidders (`?Bid`), announces the winning bid (`!Bid`) and repeats the same cycle for the next item on auction.

The session associated with the `aa` channel is similar. Since the audience do not participate in the bidding process, the behaviour of this channel is given as two cycles of `!Item.!Bid`, where the auctioneer first announces the item on auction and then announces the winning bid.

The code given in Figure 4.2 describes the Auction system where an instance of the `Auction` conversation (`auc`) is created, the items on auction are acquired and finally the participants of each role are obtained through `createAuctioneer, createBidder` and `createAudience` methods and the conversation is initiated.

Note that there is no restriction on the number of participants in each role and one can allow as many participants in each role as needed. While we have allowed multiple participants in the `Bidder` and `Audience` roles, there is only one participant in the `Auctioneer` role since this is a requirement of the example, analogous to the real-life Auction.

The code for creating an `Auctioneer` is given in Figure 4.3, where a new process is forked to join this role. This process now assumes the role of the `Auctioneer` and iterates over the list of items on auction. In each iteration, he announces the current item to both `Bidders` and the `Audience` using the `ab` and `aa` channels respectively, receives the amount each bidder is willing to pay and finally announces the maximum bid received - the winning bid - to both `Bidders` and the `Audience`.

Figure 4.4 shows the code used for creating a `Bidder` where a new process is forked to join this role. The for loop in the code corresponds to the multiple items on auction. At each iteration the process calls the `bid()` method where he communicates with the `Auctioneer`. He receives information about the item on auction, sends back the value he's willing to pay (`myBid`) for this item and is finally notified of the winning bid.

Note that the signature of the `bid()` method includes an assertion of the form `requires [(auc, d(ab)): ?Item.!Bid.?Bid ]` which expresses the

precondition of this method. It requires the current thread to have access to the `d(ab)` channel and for its behaviour to comply with the prescribed session type, i.e. `?Item.!Bid.?Bid`.

The code for creating a new member of `Audience` is shown in Figure 4.5 and can be justified similarly.

```
Conversation Auction{
  role auctioneer;
  role audience;
  role bidder;

  channel auctioneer audience aa: !Item.!Bid.!Item.!Bid;
  channel auctioneer bidder   ab: !Item.?Bid.!Bid.!Item.?Bid.!Bid;
}
```

Figure 4.1: Code for the Auction conversation

4.1.2 *Roles* Versus Contemporary Approaches

We described the auction scenario in the notation of dyadic session types in section 2.5.4. We also specified the same scenario in the syntax of global session types in section 2.6.4. In this section we compare the auction example written in the *Roles* language to its analogous description.

Note that the code given in Figure 4.1 is what we compare against the contemporary approaches since it is used to define the communication protocol of the auction system in which various roles are specified and session types are assigned to different channels. On the other hand, the code given in figures 4.2, 4.3, 4.4 and 4.5 is an example of user code stating a possible way in which the the auction system can be described. Since the *Roles* language is based on the object-oriented paradigm, the user code is written in this style. We added this code to give the reader a better understanding of the *Roles* language and further clarify some of its novel features. We do not specify the user code when considering contemporary approaches in sections 2.5.4 and 2.6.4 since they are abstract languages based on the π-calculus notation and are not associated with any programming languages.

```
Class AuctionC{
  Auction auc;  // The auction conversation

  public static void main(){
    auc = new Auction();

    Item [] items;
    items = getItems(2);
    int length = items.length;

    createAuctioneer(auc, items);

    createBidder(auc, length);
    ...
    createBidder(auc, length);

    createAudience(auc, length);
    ...
    createAudience(auc, length);

    auc.start();
  }
}
```

Figure 4.2: Code for the AuctionC class (main Auction class)

```
String createAuctioneer(Auction auc, Item [] items) requires null{
  spawn{
    auc.auctioneer.join();
    Bid winner;
    for(item in items){
      auc.aa.send(item);
      auc.ab.send(item);
      int i = 0;
      Bid [] bids;
      x = auc.ab.receive() in [if (x != EOM){
                                 bids[i] = x;
                                 i++;          } ]
      winner = max(bids);
      auc.aa.send(winner);
      auc.ab.send(winner);
    }
  }
  return ``Auctioneer created'';
}
```

Figure 4.3: Code for method createAuctioneer

```
String createBidder(Auction auc, Int length) requires null{
  spawn{
    auc.bidder.join();
    for (int i= 0 i<length; i++){
      bid();
    }
  }
  return ‘‘Bidder created’’;
}

void bid(Auction auc) requires [(auc, d(ab)) : ?Item.!Bid.?Bid ]{
  Item item;
  x = auc.d(ab).receive() in [if (x != EOM){
                                  item = x    } ]
  Bid myBid = new Bid(getMyBid());
  auc.d(ab).send(myBid);
  Bid winner;
  x = auc.d(ab).receive() in [if (x != EOM){
                                  winner = x  } ]
}
```

Figure 4.4: Code for the “createBidder” method

```
String createAudience(Auction auc, Int length) requires null{
  spawn{
    auc.audience.join();
    for (int i= 0 i<length; i++){
      listen(auc);
    }
  }
  return ‘‘Audience created’’;
}

void listen(Auction auc) requires [(auc, d(aa)) : ?Item.?Bid ]{
  Item item;
  x = auc.d(aa).receive() in [if (x != EOM){
                                  item = x    } ]
  Bid winner;
  x = auc.d(aa).receive() in [if (x != EOM){
                                  winner = x  } ]
}
```

Figure 4.5: Code for the “createAudience” method

When we compare the auction scenario described in Figure 4.1 with its analogous descriptions in dyadic session types and global session types (cf. sections 2.5.4 and 2.6.4), it is clear that the *Roles* notation is much more concise. The strength of *Roles* lies in its ability to describe multi-party systems succinctly. In Figure 4.1, the `aa` and `ab` channels characterise the communication pattern between the auctioneer and *all* members of the audience and the auctioneer and *all* bidders respectively. Since *Roles* allows us to classify different parties in a communication system as members of various *roles*, the behaviour of participants of the same role can be defined collectively to avoid repetition. For instance, in the auction example, all bidders present the same manner when communicating with the auctioneer. *Roles* allows us to gather all bidders in a role called `Bidder` and define the behaviour of the `Bidder` role with respect to other roles rather than explicitly declaring the communication pattern exhibited by each individual bidder. As a result, the auction system is specified using three roles and only two channel declarations where the number of channels is invariant to the number of participants in each role. In other words, even though *Roles* supports dynamic leaving of participants, there is no need to remove channels when the number of members in a role decrease.

On the other hand, the auction scenario when written in the syntax of dyadic session types requires dedicated channels to be defined between each member of the audience and the auctioneer as well as distinct channels between each bidder and the auctioneer. Although the expected behaviour between each member of the audience and the auctioneer is identical, we still have to explicitly *repeat* this communication pattern for each and every one of them. Therefore, we end up with a large system consisting of repetitive sessions and identical behaviours that is avoided in *Roles*. The auction example when written in the dyadic session types notation leads to (`n+m`) channel declarations where there are `n` members of the audience and `m` bidders.

Similarly, when we specify the auction system in the syntax put forward by the global session types, each participant is allocated a distinct channel over which it receives messages from all other members of the system. This approach suffers from the same shortcomings as the dyadic session types. Although each bidder exhibits the same communication pattern when interacting with the auctioneer, this behaviour must be repeated for each channel declaration leading to (`n+m+1`) channels and repetitive code. Similarly, the number of channels is dependent on the number of participants and different instances of the auction scenario must be explicitly specified since different number of participants means different number of channel declarations and

hence different systems.

The following table gives an overview of the complexity of the description for each approach assuming the presence of `n` members of the audience and `m` bidders.

Complexity \ Approach	Dyadic Session Types	Global Session Types	*Roles*
No. of channels	`n+m`	`n+m+1`	2
Other	-	–	3 roles

Figure 4.6: The complexity of different approaches when specifying the auction example.

An other advantage of the *Roles* language over the contemporary approaches lies in that when a scenario is described using conversations, it abstarcts away from the details of the scenario such as the number of attending parties and provides the user with a general template that can be used repeatedly. For instance, if the user needed to define two different instances of an auction where the number of bidders and members of the audience varies from one to the other, he can simply instantiate the same conversation twice without having to redefine any of its constituent roles or channels.

However, depending on the cardinality of the bidders or the audience, different instances of the auction example has to be modified accordingly. That is to say, if the user wants to define two different auctions with different number of bidders or the audience, he needs to specify each scenario separately since the number of channel declarations directly depends on the number of participants. This is of course a very tedious job for the user since he needs to redefine every single channel needed in each system separately and code re-using is not possible.

4.1.3 Online Book Purchase Example

In this example we specify the protocol characterising the process of buying a book from an online retailer such as Amazon. The aim of this example is to demonstrate the feature of dynamic leaving of the participants of a role.

This system consists of a single retailer and a single buyer and is defined in the `OnlineBookPurchase` conversation written in *Roles* syntax as given in Figure 4.7. We could of course assume the existence of many buyers. However, since those features of *Roles* pertaining to multiparty communication and broadcast messaging are already demonstrated in the previous example, we have limited the number of buyers to one for simplicity. This conversation consists of two roles, namely the `retailer` and the `buyer` roles as well as a channel (`rb`) between the retailer and the buyers.

Figure 4.8 shows the code for the main class where an instance of the `OnlinebookPurchase` conversation is created and new participants join the new conversation.

The session type associated with the `rb` channel asserts that first the buyer sends the title of the book and then waits for a response form the retailer about the price of the desired book. At this point, if the buyer is interested in the book at the given price he sends back his positive response to the retailer as well has his address. The retailer then informs the buyer of the estimated delivery time.

Figure 4.10 shows the code for creating a new buyer. First a new thread is spawned to join the buyer role. The new buyer then communicates the title of the book to the seller and waits for a response regarding the price of the book. Once he has received the price from the retailer, he needs to decide whether or not he is interested in the book at the given price. If he is, he sends back a positive response announcing that he is willing to buy the book and the communication continues as stated in the session type of the channel. On the other hand, if the buyer is no longer interested in the book, he simply *leaves* the conversation and the communication between the two of them ends.

The code for creating a new retailer is given in Figure 4.9. Once the new thread to assume the role of the retailer is forked, communication between the buyer and the retailer begins. He receives the title of the desired book and sends back its price. If the buyer is interested in the book at the price given, the retailer receives a positive response from him and they move on to exchange the address and the delivery date. However, if the buyer has lost interest in the book, he leaves the conversation and no further communication will take place in between them. Therefore, once the receive instruction is executed, the buyer will receive no value but `EOM` indicating that there was no messages from the buyer to retrieve and as a result, the value of the `confirm` field remains as `null`. Hence, before proceeding to the next step of communication, the buyer checks the value of the `confirm` field. In the event that it is set to null, the retailer leaves his role and the

conversation ends. Otherwise, they continue to communicate as discussed earlier.

Even though in the current version of the *Roles* language, session types do not support branching and selection, this behaviour can be replicated by making the participants dynamically leave a role. For instance, in this example the buyer is given the choice to *either* continue with purchasing the book *or* quit the system, a behaviour usually described with *selection* constructs in session types. However, we replicated this protocol by having the buyer leave the conversation if he is no longer interested in the purchase or continue shopping otherwise. On the other hand, the retailer's course of action depends on the buyer's choice, a behaviour commonly described through *branching* in session types. We similarly reproduced this behaviour through making the retailer leaving his role dynamically in the event the buyer loses interest in the book.

```
Conversation OnlineBookPurchase{
  role retailer;
  role buyer;

  channel retailer buyer rb:?Title.!Price.?Positive.?Address.!Date;
}
```

Figure 4.7: Code for the OnlineBookPurchase conversation

```
Class OnlineBookPurchaseC{
  OnlineBookPurchase obp;  // The auction conversation

  public static void main(){
    obp = new OnlineBookPurchase();

    createRetailer(obp);

    createBuyer(obp);
    ...
    createBuyer(obp);

    obp.start();
  }
}
```

Figure 4.8: Code for the OnlineBookPurchaseC class (main OnlineBookPurchase class)

```
String createRetailer(OnlineBookPurchase obp) requires null{
  spawn{
    obp.retailer.join();
    String title;
    x = obp.rb.receive() in [if (x != EOM){
                                    title = x;  } ]

    obp.rb.send(getPrice(title) );
    Positive confirm;
    x = obp.rb.receive() in [if (x != EOM){
                                    confirm = x;  } ]

    if(confirm != null){
        Address address;
        x = obp.rb.receive() in [if (x != EOM){
                                    address = x;  } ]

        obp.send(getDeliveryDate(title, address) );
    }
    else{
        obp.retailer.leave();
    } in the style of declarative programming
  }
  return ``Retailer created'';
}
```

Figure 4.9: Code for method createRetailer

```
String createBuyer(OnlineBookPurchase obp) requires null{
  spawn{
    obp.buyer.join();
    obp.d(rb).send(getTitle() );
    Price price;

    x = obp.d(rb).receive() in [if (x != EOM){
                                    price = x  } ]

    if(price <= budget){
        buy();
    }
    else{
        obp.buyer.leave();
    }
  }
  return ``Buyer created'';
}

void buy() requires [(obp, d(rb)) :!Positive.!Address.?Date ]{
  Positive pos = new Positive();
  obp.d(rb).send(pos);
  obp.d(rb).send(getMyAddress() );

  Date deliveryDate;
  x = obp.d(rb).receive() in [if (x != EOM){
                                   deliveryDate = x    } ]
}
```

Figure 4.10: Code for the "createBuyer" method

4.2 Operational Semantics

Global session types as discussed in chapter 2 represent the nearest point of comparison for our work on *Roles*, and we will therefore attempt to evaluate our approach to operational semantics against theirs.

One fundamental difference between our semantics and that of global session types lies in the rewriting rules of the communication actions. For instance, the semantics of the receive action in the global session types is as follows.

$$\mathtt{s?(x);P} \mid \mathtt{s} : \mathtt{v}.\overline{\mathtt{h}} \longrightarrow \mathtt{P[v/x]} \mid \mathtt{s} : \overline{\mathtt{h}}$$

In global session types, message queues for different channels are treated as processes and run in parallel with communication actions. In the above rule, $\mathtt{s} : \mathtt{v}.\overline{\mathtt{h}}$ is a message queue for the `s` channel. A receive instruction over a channel can only be reduced if it is running in parallel with the message queue of the channel, in which case the desired message is retrieved from the queue and the modified queue is returned.

However, in our approach the concept of message queues is replaced by noticeboards shared amongst participants of a role. Noticeboards are created upon initiation of a conversation and hence in the semantics of the receive action we can guarantee their presence in the heap when a receive instruction is executed.

One further difference between the semantics of *Roles* and the global session types is of that each approach includes rules which the other lacks. We have introduced new instructions to the syntax of *Roles* such as `join`, `leave` and `start` to enable dynamic leaving of participants as discussed in depth earlier. Although we have expanded the rewriting rules of *Roles* in comparison to that of global session types, we believe this is for good reasons since it provides the user with a more flexible language where advanced features such as dynamic leaving is supported.

On the other hand, the global session types include rewriting rules for session delegation and branching. Unfortunately, including these features in *Roles* was beyond the timescale of the project. We include these in further work in section 5.2.

4.3 Type System

In this section we evaluate the type system of the *Roles* language. As in previous section, we compare our type system against that of the global session types discussed in chapter 2.

4.3.1 Type Safety

A type system is *sound* if it can be shown that no well-typed program can cause a typing error. We discuss the implications of this in section 2.2 when we discuss common communication bugs and the manner in which typing could potentially solve them. These are examples of situations that any well-typed program written in the *Roles* language should never encounter, if the type system is sound.

In section 3.6 we introduced various definitions leading to our conjecture that the type system of the *Roles* language is sound through the subject reduction conjecture (cf. 3.6.6). Proof of the subject reduction property was, unfortunately, outside of the timescale of the project. However, this forms a part of our further work which is discussed in section 5.2.

Here again, we compare the type system of the *Roles* language with that of the global session types [32]. One of the major differences between the type system of the two approaches lies in the runtime typing rule for the message queues. Since message queues are considered as processes in the global session types, they need to be typed at runtime.

Suppose we have the following threads running in parallel where process `p` is sending the value `true` over channel `s` followed by number `3` as the next value. Process `q` is expecting to receive two values on channel `s` and $\mathtt{s} : \emptyset$ represents an empty message queue for channel `s`.

$$\mathtt{s}! < \mathtt{true} >; \mathtt{s}! < 7 >; 0 \mid \mathtt{s} : \emptyset \mid \mathtt{s}?(\mathtt{x}); \mathtt{s}?(\mathtt{y}); 0$$

After one step of reduction we have the following processes.

$$\mathtt{s}! < 7 >; 0 \mid \mathtt{s} : \mathtt{true} \mid \mathtt{s}?(\mathtt{x}); \mathtt{s}?(\mathtt{y}); 0$$

The message queue for channel `s` is typed as follows where [ ] indicates a hole since processes `p` and `q` have not finished interaction over channel s.

$$\mathtt{s} : \{! < \mathtt{bool} >; [\,]@\mathtt{p},\ [\,]@\mathtt{q}\}$$

Finally, after a few steps of reductions when there are no possible reductions are left, the message queue for channel s can be typed as follows.

$$\texttt{s} : \{! < \texttt{bool} >; ! < \texttt{int} >; \texttt{end@p},\ ? < \texttt{bool} >; ? < \texttt{int} >; \texttt{end@q}\}$$

The runtime typing rule for a message queue is then given below, denoting that if a process `p` has dropped a message of type `U` in message queue of channel `k` and process `q` has received a message of type `U` from the message queue of channel `k`, then message exchanging has taken place between `p` and `q` and the type of the message queue can be modified accordingly.

$$\texttt{k}! < \texttt{U} >; \texttt{H@p}, \texttt{k}? < \texttt{U} >; \texttt{T@q} \xrightarrow{\texttt{k}} \texttt{H@p}, \texttt{T@q} \qquad [\texttt{TR} - \texttt{Com}]$$

The notion of typing message queues in the global session types is analogous to our notion of runtime session environment, Σ, when typing the communication actions, `send` and `receive`. When typing a process at runtime, the initial session environment (Σ) is modified depending on the expression being executed producing the final session environment (Σ').

However, unlike the typing rule for message queues in global session types, in our approach when we type a `send` or `receive` action, we *only* inspect the session type assigned to the current process for the desired channel. For instance, when we type an expression of the form $\texttt{x} = \kappa.\texttt{ch.receive}()$ `in e`, we only inspect the session type of the (κ, `ch`) pair in Σ. We do not check the session type of (κ, `d(ch)`). When a conversation starts, the session environment is updated with the session types of all of the channels of the conversation and their duals. Hence, if as a participant of a role we expect to receive a message of type `t` and this is reflected in Σ, we know that the participants of the opposite role are also bound by Σ to show the dual behaviour and send a value of type `t`. Hence we skip checking the session type of the dual channel and seek to retrieve the message from the notice board. The message we expect to receive is either available in the notice board or the participants of the dual role have not finished sending it in which case we will have to wait. Either way, we know that the type of the message we receive will agree with the type we are expecting to receive.

Therefore, we believe our type system offers stronger guarantees in terms of type safety.

4.3.2 Progress

In section 3.7 we conjecture that the *Roles* language has the progress property. However, when compared to the contemporary approaches such as the global session types, the *Roles* language makes weaker promises when it comes to the progress property. Our promise is that a process `P` is alive

so long as it is well-typed; that is, it can be typed by the typing rules in section 3.5.10. However, the weakness of our guarantee lies in the definition of liveness given in section 3.7. There are several ways for a process to get stuck and hence not be alive anymore.

For instance, if a process is exucuting an expression `e` which involves starting a conversation that has already started, it gets stuck. Unfortunately, we cannot pre-empt this situation since it cannot be detected at compile time. We cannot statically check if a conversation has been initiated since it depends on the flow of the program and can be only determined at runtime.

A similar case arises when an expression involves joining a conversation that has already started. As discussed above, the status of a conversation cannot be detected at compile time and therefore we cannot statically detect such expressions.

A final case in which an expression `e` can get stuck is by trying to send or receive a value over a channel of a conversation that has not yet started. Although the reasons why `e` gets stuck in this case are similar to those mentioned in the previous cases, the kind of block `e` faces in this case can be distinguished from the former ones. If `e` gets blocked due to the reasons discussed in the last two cases, it will be *permanently* blocked and it will never recover. However, if `e` gets stuck while trying to communicate over a channel of a conversation that has not yet been initiated, there is a chance for `e` to recover. At any point, if the conversation in question is started, `e` can resume communicating over its channels. In a sense, `e` can be seen as *busy waiting* while the conversation is not initiated. Of course if the conversation is never started, `e` will be blocked forever.

4.4 Challenges

In this section we discuss some of the major difficulties and obstacles encountered in the design of the *Roles* language, both in terms of design decisions and complexities with our approach. We discuss how we resolved them, and justify our solutions.

4.4.1 Learning Curve

As with any project of such a deeply theoretical nature, there is an initial learning curve that must be passed in order to cover significant ground. The breadth of work that has already been done in the field of session types is considerable. Making sense of the many diverse approaches to the topic was not simple, and required us to work to understand a variety of different notations, definitions and ways of working. In addition, the *Roles* language represents a novel approach to the concept of session types. As such, our work challenged us to create new solutions to problems not encountered in other approaches, including those that we describe in the remainder of this section.

4.4.2 Refactoring and Refinement

When designing the *Roles* language and specifying its operational semantics we went through several refactoring phases. There were multiple occasions where we had to retract and reconsider the design decisions we had made either due to inconsistencies in the end result or in order to better the system at hand. We have presented a list of design specific desicisons that we revisited and revised.

Participants of Roles

We initially committed to populate the roles with objects and ask the user to provide us with an expression to be executed upon joining. Our original syntax for joining a conversation was `cv.r.join(o, e)` where `o` referred to the object to join role `r` of conversation `cv` and `e` was the expression to be run once the join instruction was completed.

At later stages we encountered serious problems regarding the safety of communication actions. For instance, when the send command `cv.ch.send(v)` was to be executed, we could not determine the identity of the participant intiating this action. In other words, whenever a thread reached a send action, we could not identify the participant and his attending role who was in

charge of the send action. As a result, we could not guarantee the legality of any of the communication actions since we had no mechanism in place to separate rogue instructions from those permitted and hence we could not authorise them.

Of course there were quick fixes available. For instance, we could add an additional argument to each communication action denoting the identity of the object that was behind the action. However, this was a highly inelegant approach and would have decreased the readability of *Roles*. Therefore, we decided to populate the roles with process identifiers and determine their access rights to channels depending on the roles they attend. This way, whenever a send instruction is reached, we can check the identity of the currently running thread against those permitted to communicate over the channel and hence authorise valid instructions only.

Moreover, we ommitted the expression that determined what to be executed next from the syntax of the `join` instruction, since it led to redundant information in the heap in the syntax of `join` that could be easily pre-empted. We took a more natural approach and had the subsequent expressions after the `join` statement to be executed upon completion of next.

The Judgement of Reduction Rules

Altering the way we populate the roles as discussed above, resulted in modifying the signature of the rewriting rules. We had to include the identity of the currently active thread (π) in the signature to extract various information from the heap. For instance, each participant keeps track of several indices in the heap to access different channels and communicate messages accordingly. This indexing information is organised in the heap by means of mapping each process identifier to the relevant set of indices. Throughout execution, whenever a `send` (`receive`) instruction is reached, the indices for the current thread are looked up in the heap to determine the destination (source) of the message. To accomplish this, we equipped each reduction rule with the identifier of the current process (π).

Role Declaration

In the previous versions of our specification, the syntax of *Roles* required each role to be assigned a class type upon declartion. This was mainly to share common behaviours and methods between participants of a role since we had originally committed to populating roles with objects. However, as discussed in the 'Participants of Roles' section, we later decided to populate

roles by threads. Therefore, associating each role with a class type did no longer benefited us and after further inspection, we altered the syntax so that the declaration of roles consists of role identifiers only.

Spawning New Threads

In the earlier versions of the *Roles* language, we did not have the `spawn` instruction included in the syntax. New threads could not be spawned by the user and whenever a join instruction was reached, the system would fork a new thread in the background and add it to the desired role.

This approach came with its faults and restrictions. If we spawned a new thread every time a new participant was to join a role, the user would not be aware of the identity of the new process and thus could not ask for a particular participant to leave a role at later stages. This could be fixed by returning the identity of the new thread to the user at the price of losing its readability. In large systems with many participants, the user would have to keep track of the identity of each and every participant, something that the user rightfully expects to be taken care of by the system.

In addition to its inelegance, this approach suffers from other limitations. For instance, scenarios in which a participant attends multiple roles cannot be reproduced in this approach since each join instruction results in creating a distinct new thread and one thread cannot be a participant of many roles.

Given the restrictions discussed above, we abandoned this approach and sought alternatives. In the final revision of the *Roles* language, the user is in charge of spawning new processes as required.

Initiation of a Conversationn

Originally, we did not have the `start` instruction included in the syntax of the *Roles* language. A conversation would initiate upon declaration and since we did not allow for new members to join the roles of an ongoing conversation, conversation constructors had additional arguments denoting the participants of each role. It was soon evident that this approach was unwieldy as it reduced both the readability and conciseness of the language by loading so much information in the conversation constructors.

Thus, we included the start instruction in the syntax so that the user can explicitly initiate a conversation when needed. New participants can join the roles of a conversation prior to its initiation.

Type System and the Receive Instruction

At later stages of designing the *Roles* language, while specifying its type system, we discovered a fatal problem with our specification that we had formerly overlooked.

Previously, the syntax of the `receive` instruction was the mirror of the `send` action where we would write `cv.ch.receive()` to receive the next value on channel `ch` of conversation `cv`. This instruction would return a single message at a time; consecutive `receive` instructions were required to retrive *all* messages in the current receiving round and an EOM message would finally be returned to mark the end of the current round.

However, when we began work on the type system, we failed to correctly type the `receive` instruction. Since the number of participants in a role is not statically known, the type system cannot determine when to advance the session environment. For instance, suppose for initial session environment Σ we have:

$$\Sigma(\texttt{cv}, \texttt{ch}) = ?\texttt{Char}.!\texttt{Int}$$

and that we are type checking the code shown in Figure 4.11 at compile time.

```
...
cv.ch.receive();
cv.ch.send(7);
...
```

Figure 4.11: This code could not be correctly type-checked in earlier versions of *Roles*

Suppose that channel `ch` is defined between the $\texttt{r}_1$ and $\texttt{r}_2$ roles and that the current thread is a participant of the $\texttt{r}_1$ role. We now assume two different scenarios.

First, assume there is only one participant in the $\texttt{r}_2$ role. When the `receive` statment in the code is reached, we receive a single message from the participants of the $\texttt{r}_2$ role. Since there is only one member in the $\texttt{r}_2$ role, there are no further messages to be retrieved and hence this receiving round is complete. Therefore, when typing the `receive` statement, it is safe to modify the session environment Σ so that:

$$\Sigma(\texttt{cv}, \texttt{ch}) = !\texttt{Int}.$$

On the other hand, assume that there are several participants in the $\texttt{r}_2$ role. When the `receive` statement in the code is reached, we receive a single message from the participants of the $\texttt{r}_2$ role. However, since there is more than one member in the $\texttt{r}_2$ role, there are still more messages to be retrieved and hence this receiving round is *not* complete. Therefore, when typing the `receive` statement, the `?Char` part of the session has not yet been fully consumed and we *cannot* change Σ to reflect:

$$\Sigma(\texttt{cv}, \texttt{ch}) = !\texttt{Int}$$

As demonstrated in the scenarios discussed above, if we allow the `receive` instruction to mirror the `send` action and retrieve one message at a time, we cannot safely type check our language. Thus, in the final version of the *Roles* language, the `receive` statement is only allowed to be used in a particular format. We use the following syntax to receive messages.

$$\texttt{x} = \texttt{cv.ch.receive}() \texttt{ in e}$$

As specified by the semantics of the receive rule in section 3.4, this instruction acts as a loop and consecutively retrieves the messages communicated in the current receiving round and substitutes it for `x` in the expression `e`. When all messages in the round are retrieved, an `EOM` message is received to indicate the end of the current round. When a `receive` instruction is typed by the typing rules described in section 3.5.8, it is safe to advance the initial session environment (Σ), since we know at the end of this instruction all messages in the current receiving round are retrieved.

This issue would not have arisen in other contemporary approaches discussed in chapter 2, since they do not make use of broadcast communication and so the concept of retrieving multiple messages is irrelevant.

Method Signature

Originally, we had defined the signature of method declarations in *Roles* similar to that of standard object-oriented languages; that is, without the session environment included. However, at later stages while working on the type system of *Roles*, we discovered a problem that proved the type system of *Roles* unsafe.

Suppose we have the code shown in Figure 4.12. When executing the method `main` in class `C1`, two instances of the conversation `CV`, namely `cv1` and `cv2` are created. The current thread then joins the role `r1` of `cv1`, `cv1` is started and the `m1` method is called. In the body of the `m1` method, we

first execute the `cv1.ch.send(7)` instruction which is permissible since the current thread is a member of the `r1` role in conversation `cv1` and the session type of `ch` allows the participants of role `r1` to send an integer.

On the other hand, execution of `cv2.ch.send(2)` is not allowed since the current thread is *not* a member of role `r1` in conversation `cv2`. Moreover, the `cv2` conversation has not started yet, hence no communication is allowed over its channels. However, this was overlooked in our older designs and such illegal actions were permitted.

Therefore, we modified the syntax of method declarations in *Roles* so that it includes a session environment asserting the precondition of the method. In other words, this session type asserts the kind of behaviour it expects from the threads executing its body. If the precondition is not fulfilled, the execution of the method is not permitted and hence illegal actions can be pre-empted.

Conclusions

In this chapter we evaluated and summarised our work on *Roles*. We have discussed our work on the syntax, semantics and type system and stepped through the strengths and weaknesses of our approach. In particular, we compared our approach with those described in chapter 2 and demonstrated the benefits of the *Roles* language. Finally, we discussed the difficulties met in designing this language, and the challenges we overcame in order to produce the version presented in this report. In the next chapter we summarise our work and look to the future development of the language.

```
conversation CV{
  role r1;
  role r2;

  channel ch r1 r2 : !Int.?Char
}


public class C{
  void m1(){
      cv1.ch.send(7);
      cv2.ch.send(2);
  }
  public static void main(){
    CV cv1 = new CV();
    CV cv2 = new CV();

    cv1.r1.join();
    cv1.start();
    m1();
  }
}
```

Figure 4.12: An example code showing the necessity of session environments in method declarations.

Chapter 5

Conclusion

In this section we conclude our work; summarising *Roles* and its major achievements, outlining further work and extensions to develop the language and finally reflecting on the project and its future evolution.

5.1 Achievements

In this project we introduced a new language, *Roles*, designed to describe the communication scenarios between groups of participants. In chapter 3, we specified the syntax of *Roles*, described its operational semantics and presented its type system. We then made conjectures about the properties of the type system, in particular those pertaining to type safety and progress of programs written in the language of *Roles*.

The language of *Roles* was developed while bearing the shortcomings of existing approaches in mind. Global session types do not allow for multiparty communication in which the number of participants dynamically change. On the other hand, while dyadic session types cater for dynamic channel creation, they suffer from inevitable duplication while defining communication scenarios with multiple peers (cf. discussion in section 2.5.5). Therefore, in *Roles* for the participants to dynamically leave a role during an ongoing conversation.

Another strength of the *Roles* language is its simplicity for broadcast communication. Since the channels are between roles rather than individual participants, communication is by definition broadcast. When sending a message over a channel, *all* participants of the role at the receiving end-point will receive the communicated message and therefore, message broadcast is the default form of communication.

Both dyadic and global session types allow for giving distinct behaviours to each of the participants since the session type is defined per channel and therefore per participant. Whereas in the *Roles* notation, the behaviour of *all* members of a role is given in a single session type as prescribed by the channel the said role communicates over. This behaviour can be replicated in *Roles* by introducing new roles into the conversation. In other words, whenever a different behaviour is expected from one of the participants, a new role should be created and required channels between this role and the existing ones must be defined with the anticipated session type attached to it.

We believe that the *Roles* notation is at least as expressive as dyadic and global session types while showing more flexibility for representing large systems with complex multiparty interaction scenarios.

5.2 Further Work

In this report we have presented a groundwork for the language of *Roles*, defining the syntax, operational semantics and the type system. In this section we discuss the next steps that must be taken in order to extend, strengthen and better define the language and its properties. While the language in its present state can be reasoned about and evaluated at length, the further work we list below is integral to bringing the *Roles* language to a level where it can be compared with its contemporaries.

5.2.1 Session Types as Primitive Types

In the current version of the *Roles* language, session types are treated as special types used in channel declarations in order to describe the communication protocol of the said channel. In other words, session types are not considered as primary types used in field or method declarations; we cannot instantiate fields of this type or use them in method signatures as argument or return types.

The domain of the recognised types by the *Roles* language consists of the classes and conversations only `(t = C | CV)` as presented in Figure 3.1. We would like to expand this to include session types as primary types where they can be used to declare the type of fields or methods. This way, not only we increase the readability of the language, we can also use the session types as arguments to conversation constructors where the user can create to instances of the same conversation with different behaviours if he sees fit.

5.2.2 Conversations as Communicated Types

The notion of delegation as discussed in section 2.4 is irrelevant in the context of the *Roles* language since communication channels are only accessible by those members who participate in the relevant roles. If we communicate channels as messages, the recipient cannot use the received channel unless he already has access rights to the channel in question through role membership. Nevertheless, one can replicate similar behaviour as the higher order sessions by communicating *conversations* as messages. Once a conversation instance is received, the recipient can join any of its roles and start communicating so long as the conversation has not yet started or dynamically joining a role is permitted (Cf. 5.2.4).

Our current version of the *Roles* language does not support higher order sessions. Session types consist of ordinary communication actions such as input (?) and output (!) where the type of the communicated message ranges over the `classes` only. However, one can extend *Roles* by allowing conversation types to be communicated over channels which will in turn provide user with higher order sessions as discussed above.

5.2.3 Signature of Sessions

In our current version of the *Roles* language, the session types used to describe the behaviour of communication channels allow for primary communication actions, namely input (?) and output (!), only. However, it is highly desireable for communication systems such as *Roles* to provide the user with higher level session constructs such as recursion, branching / selection and session variables. Session variables are particularly useful when session types can be used as primary types as discussed in section 5.2.1. Once, the *Roles* language is equipped with such powerful constructs, many other use cases and common examples from the literature can be specified in our notation. We have written a few examples in *Roles* syntax in Appendix A where we have assumed the existence of session constructs such as recursion and session variables in *Roles*.

5.2.4 Dynamic Joining

One of the strengths of the current version of the *Roles* language lies in its suport for dynamic leaving. That is, participants of roles can end their membership in a role at any point during the conversation whether or not the conversation has already started. When a member leaves its role (r_1), if the participants of the opposite role (r_2) were to receive messages from

members of r_1, they no longer expect to receive as many messages since the cardinality of r_1 has changed and they will move on once they receive messages from all other participants.

However, we can take this one step further by allowing participants to dynamically *join* a conversation. In other words, one can extend *Roles* by allowing new members to join any role at any point during the conversation whether or not it has already been initiated. Inclusion of this feature requires thorough investigation since several design alternatives are available and design decisions are to be taken carefully.

For instance, when a new member joins a conversation, how do we define the sessions associated with the channels it can use to communnicate? The first answer that springs to mind is to start each channel with its initial session type defined at source level. Nevertheless, this can clash with the behaviour of other participants since they might have already consumed the session past the initial declaration by performing the expected communication actions in which case the new member will fail to advance and hence get blocked forever.

Another possibility is to inspect the current members of the role and for each channel find the *slowest* participant, that is the one who has consumed the least of the initial session compared to other participants. We can then initialise the new member by associating each of its channels with the session type of the slowest participant over that channel. Nonetheless, this approach also suffers from inconsistencies. For instance, the new member will not be aware of the past interactions. Suppose that the advancement of the conversation depends on the decisions the participants make (e.g. a consensus); in the event that participants base their decisions on knowledge obtained through previous interactions, the new member will have no information available and hence can get blocked forever, in doing so, can prevent the entire system from progression.

There are ways of resolving this problem, however. One can provide the new member with a *history* of previous communications or can assert preconditions on methods which allow for new members to dynamically join a role. This was beyond the scope of this project but we hope to devise a robust solution in the future.

5.2.5 Establishing Properties of the Type System

In sections 3.6 and 3.7, we discussed various properties of the type system of *Roles*. In particular, we conjectured that the type system of the *Roles* language benefits from both *soundness* and *progress* properties. In future

work we hope to establish these properties by providing proofs for each one of these properties as well as proofs for the lemmas subsequently employed to validate our results.

5.2.6 Towards a Distributed System

Our work currently defines participants as threads, all operating concurrently on the *same* machine. The motivation for a session type language, however, is to secure systems of arbitrary size and complexity. In particular, further work on the language must broaden the *Roles* language so that it can be applied to distributed systems, where participants of different roles can be potentially running on physically *separate* machines, communicating through a complex network. For large and unreliable networks such as the Internet, this will require us to focus more on the robustness and security of our language, and demand that we prove more properties of the language.

A variant of the auction example is given in appendix A to give the reader a flavour of the end result once the further work discussed in this section is undertaken in particular those discussed in sections 5.2.3 and 5.2.4.

5.3 Reflection

At the start of this report we discussed the importance of reliable and safe communication in modern computing. However, the motivation for *Roles* is not just the problem of safe communication, but also the shortcomings of existing attempts to solve that problem. The process of design and refinement outlined in this report came from a desire to create a solution that was flexible and expressive, not merely functional.

Of course, much work remains to be done to bring *Roles* to a level where it can be widely used, as we have discussed in the previous section. We outline such further work in section 5.2 because we believe *Roles* is capable of developing into something genuinely useful to programmers. Our motivation for designing within the object-oriented paradigm was precisely this - so that we would bear in mind at all times the restrictions of implementation, and design something that could be built upon an existing framework language, such as Java. We believe that with *Roles* we have laid foundations that can used be without learning large volumes of new theory, whilst being flexible enough not to restrict the user in what they can express, and hope that an implementation will be possible in the near future.

Bibliography

[1] `http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm`.

[2] Web services choreography working group. web services choreography descriptio language.

[3] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, New York, NY, USA, 1997. ACM.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[5] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005.

[6] Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. Typechecking safe process synchronization. *Electr. Notes Theor. Comput. Sci.*, 138(1):3–22, 2005.

[7] L. Caires and H. T. Vieira. Conversation types. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, Lecture Notes in Computer Science. Springer-Verlag, 03 2009.

[8] Marco Carbone, Kohei Honda, and Nobuko Yoshida. A calculus of global interaction based on session types. *Electron. Notes Theor. Comput. Sci.*, 171(3):127–151, 2007.

[9] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. pages 2–17. Springer, 2007.

[10] Marco Carbone, Kohei Honda Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-talbot. A theoretical basis of communication-centred concurrent programming. Technical report, 2006.

[11] Curtis Clifto. Concurrency in the curriculum: Demands and challenges. *First Workshop on Curriculum in Concurrency and Parallelism*, 2009.

[12] William Cook and Jayadev Misra. Structured interacting computations. pages 139–145, 2008.

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.

[14] D. E. Culler and G. K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 60–69, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[15] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer-Verlag, 2007.

[16] Mariangiola Dezani-ciancaglini, Ugo De Liguoro, and Nobuko Yoshida. On progress for structured communications. In *In TGC07, LNCS*. Springer, 2007.

[17] Mariangiola Dezani-ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *In Proceedings of ECOOP06, LNCS*, pages 328–352. Springer, 2006.

[18] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. I_{doos}: a Distributed Object-Oriented language with Session types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC 2005*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[19] Mariangiola Dezani-ciancaglini, Nobuko Yoshida, Alexander Ahern, Er Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *In Symposium on Trustworthy Global Computing, LNCS*, pages 299–318. Springer, 2005.

[20] Professor Sophia Drossopoulou. L2 - a formal, minimal, imperative, class based, object-oriented language with inheritance, without overloading.

[21] Sophia Drossopoulou, Dezani Dezani-Ciancaglini, and Mario Coppo. Amalgamating the Session Types and the Object Oriented Programming Paradigms. In *Multiparadigm Programming with Object-Oriented Languages 2007 (an ECOOP workshop)*, July 2007.

[22] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[23] Dennis Gannon and Dan Reed. Parallelism and the cloud.

[24] Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. Bass: boxed ambients with safe sessions. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 61–72, New York, NY, USA, 2006. ACM.

[25] Simon Gay, Vasco Vasconcelos, Antonio Ravara, Simon Gay, Vasco Vasconcelos, and Antnio Ravara. Session types for inter-process communication, 2003.

[26] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 74–90, London, UK, 1999. Springer-Verlag.

[27] Elena Giachino, Matthew Sackman, Sophia Drossopoulou, and Susan Eisenbach. Softly safely spoken: Role playing for session types. In *PLACES '09*, 2009. To appear.

[28] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[29] Kohei Honda. Types for dyadic interaction, 1993.

[30] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

[31] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web services, mobile processes and types. *EATCS Bulletin*, 91:160–188, 2007.

[32] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, 2008.

[33] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal techniques for Distributed Systems; Proceedings of FMOODS/FORTE*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[35] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. *SIGPLAN Not.*, 28(7):54–63, 1993.

[36] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. *SC Conference*, 0:37, 1995.

[37] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[38] Robin Milner. Functions as processes. In *ICALP '90: Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 167–180, London, UK, 1990. Springer-Verlag.

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

[40] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[41] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, 1992.

[42] Marius Nagy and Selim G. Akl. On the importance of parallelism for quantum computation and the concept of a universal computer. In *Proceedings of the Fourth International Conference on Unconventional Computation*, pages 176–190, 2005.

[43] Charlene O'Hanlon. A conversation with steve ross-talbot. *Queue*, 4(2):14–23, 2006.

[44] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, New York, NY, USA, 2006. ACM.

[45] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. *Pac Symp Biocomput*, pages 459–470, 2001.

[46] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, Imperial College London, June 2008.

[47] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[48] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[49] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE '94: Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 398–413, London, UK, 1994. Springer-Verlag.

[50] Antonio Vallecillo, Vasco T. Vasconcelos, and Antnio Ravara. Typing the behavior of objects and components using session types, 2002.

[51] Vasco Vasconcelos, Simon Gay, and Antonio Ravara. Session types for functional multithreading. In *In CONCUR04, volume 3170 of LNCS*, pages 497–511. Springer-Verlag, 2004.

[52] Vasco T. 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.

[53] H. T. Vieira, L. Caires, and J. C. 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*, Lecture Notes in Computer Science. Springer-Verlag, 03 2008.

[54] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, 1990.

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

Appendix A

Example Scenarios for *Roles* Language Evaluation

Auction Example with *Roles*

In this example we describe the protocol defining the auction communication system in *Roles* syntax. We assume that the syntax of *Roles* has been extended to accommodate advanced features listed in section 5.2.

The `AdvancedAuction` protocol has been described in Figure A.1. There are several items on sale and the auction consists of multiple rounds - one round per item - denoted by the recursion variable X. Each round of the auction is timed and at each round the auctioneer first announces the item on auction. He then repeatedly listens for bids from bidders interested in the item (recursion variable T) and announces the highest bid so far (recursion variable Y).

The code for initialising the auction system is given in Figure A.2 where an instance of the `AdvancedAuction` conversation is created and the methods `createAuctioneer` and `createAudience` are called to generate new participants for the `auctioneer` and `audience` roles, respectively.

Figure A.3 gives the code for creating a new member of the `auctioneer` role. First, a new thread is spawned to join the `auctioneer` role. The new thread (auctioneer) then iterates through the list of items on auction. At each round, he announces the item and its starting bid which includes the reserved price and indicates that the auction is ongoing. Henceforth, he

repeatedly checks to see if the timeout for the current item has reached; if there is still time remaining, he receives more bids from the bidders, calculates the highest bidder and if the highest bid is different from the last time, he announces it to the audience. On the other hand, if the timeout is reached, he he announces the winner of the item and indicates the end of auction for the current item.

The code for creating a member of the audience is given in Figure A.4. After a new thread is forked and the new participant has joined the `audience` role, the new member then enters consecutive rounds of auction for various items on sale. At each round he first receives the item description and its starting bid from the auctioneer. He then repeatedly checks if the auction has ended; if there is still time for further bids and he is interested in the item, he becomes a bidder by *joining* the `bidder` role. He sends his bid to the auctioneer and *leaves* the `bidder` role.

```
conversation AdvancedAuction{
  role auctioneer;
  role audience;
  role bidder;
  channel auctioneer audience aa :μX.!Item.!Result.μY.!Result.Y.X;
  channel auctioneer bidder   ab :μT.?Bid.T;
}
```

Figure A.1: Code for the AdvancedAuction conversation

```
Class AdvancedAuctionC{
  AdvancedAuction auc;

  public static void main(){
    auc = new Auction();

    Item [] items;
    items = getItems();
    int length = items.length;

    createAuctioneer(auc, items);

    createAudience(auc, length);
    ...
    createAudience(auc, length);

    auc.start();
  }
}
```

Figure A.2: Code for the AdvancedAuctionC class (main AdvancedAuction class)

```
String createAuctioneer(AdvancedAuction auc, Item [] items) requires null{
  spawn{
    auc.auctioneer.join();
    Result winner;
    Bid currentBid;
    Bid [] bids;

    for(item in items){

      auc.aa.send(item);
      winner = new Result(reservedBid, false);
      auc.aa.send(winner);

      while(!isFinished() ){
        x = auc.ab.receive() in [if (x != EOM){
                                        bids[i] = x; } ]
        Bid newBid = max(bids);
        if (newBid > currentBid){
          winner = new Result(newBid, false);
          auc.aa.send(winner);
        }
      }
      winner = new Result(currentBid, true);
      auc.aa.send(winner);
    }
  }
  return ``Auctioneer created'';
}
```

Figure A.3: Code for method createAuctioneer

```
String createAudience(AdvancedAuction auc, Int length) requires null{
  spawn{
    auc.audience.join();
    Bid bid;
    for(int i = 0; i< length; i++){
      Item item;
      x = auc.d(aa).receive() in [if (x != EOM){
                                        item = x; } ]
      while(true){
        x = auc.d(aa).receive() in [if (x != EOM){
                                        bid = x;
                                                  } ]
        if (bid.done){
          break;
        }
        else if (bid.pid != self && interested){
          auc.bidder.join();
          auc.d(ab).send(getMyBid() );
          auc.bidder.leave();
        }
      }
    }
  }
  return ``Audience member created'';
}
```

Figure A.4: Code for method createAudience

English Auction Example

An English auction, also called an *open ascending price auction* because of the properties it has, is the most common form of auction. The auctioneer may announce prices, bidders in turn may bid *openly* against each other. A successful bid is one higher than the existing winning bid. When no-one is willing to make a higher bid, the auction ends and the winning bidder pays the value of his bid. One exception to this is if the winning bid is lower than the *reserve price* set by the seller. If the winning bid is lower than this price, it will not be sold.

The protocol defining the auction communication system is given below in *Roles* syntax.

`conversation` *EnglishAuction* {
  `role` *Auctioneer*
  `role` *Attendants*
  `role` *Bidders*
  `channel` *Auctioneer Attendants* $k1$:$!Item.!Price$
  `channel` *Bidders Attendants* $k2 : \mu T.!Price.T$
}

The *Attendant* role includes the bidders, the auctioneer and everyone else present at the auction. At the start of the auction, the Auctioneer announces the item on sale and its reserved price to the attendants (bidders and audience) over the $k1$ channel. At this point, the bidders constantly bid for the item announcing the amount of their bid to the auctioneer, other bidders and the audience. They do so by sending the bid value over the $k2$ channel addressed at the attendants.

Program Committee Meeting Example

When an *author* submits a paper to a conference it must first pass a Programme Committee who will decide whether or not to accept it. Such a committee is composed of *members*, of which one is the *leader* of the committee also. When the committee receives a paper, the members vote on whether or not to accept the paper.

If the vote results in a majority decision, the leader contacts the author and informs them of the decision. If no majority exists and the vote is drawn, the leader asks the members to re-consider the paper and vote again in order to produce a majority. She will do this subsequent times until a decision is reached.

`conversation` $ProgramCommitteeMeeting$ {
  `role` $Author$
  `role` $ComMember$
  `role` $ComLeader$
  `channel` $ComLeader\ ComMember\ k1 : \mu T.!Paper.\mu X.?Vote.!Result.X.T$
  `channel` $ComLeader\ Author\ k2 : \mu T.?Paper.!Result.T$
}

The behaviour of channel k between the committee leader and the committee members can be described as follows. The leader constantly introduces new papers to the committee members and asks them to vote. This is described with the recursion variable T. Once the committee members are informed of the paper, voting starts and in the event of not reaching a majority decision, the voting process is repeated. This is given as the recursion variable X.

Furthermore, the committee leader communicates with the paper authors over the $k2$ channel. The committee leader constantly listens for new paper proposals from the authors and notifies them of the decision made regarding the acceptance of the paper.

Appendix B

Operational Semantics of *Roles*

Process

$$\frac{\mathsf{P}_1,\ \chi \rightsquigarrow\ \overline{\mathsf{P}_2},\ \chi'}{\overline{\mathsf{P}_a} \mid \mathsf{P}_1 \mid \overline{\mathsf{P}_b}\ ,\ \chi \rightsquigarrow \overline{\mathsf{P}_a} \mid \overline{\mathsf{P}_2} \mid \overline{\mathsf{P}_b}\ ,\ \chi'}$$

Context

$$\frac{(\mathsf{e}, \pi), \chi \rightsquigarrow (\mathsf{e}', \pi) | \overline{\mathsf{P}}, \chi'}{(\mathsf{Ctxt}[\mathsf{e}], \pi), \chi \rightsquigarrow (\mathsf{Ctxt}[\mathsf{e}'], \pi) | \overline{\mathsf{P}}, \chi'}$$

Spawn

$$\frac{\pi' \notin \chi \quad \chi' = \chi[(\pi, \mathsf{null}, \mathsf{null}) \mapsto (0,0,0)]}{(\mathsf{spawn}\{\mathsf{e}\}, \pi), \chi \rightsquigarrow (\mathsf{null}, \pi) | (\mathsf{e}, \pi'), \chi'}$$

NewC

$$\frac{\mathsf{FS}(\mathsf{C}) = \overline{\mathsf{C}\ \mathsf{f}} \quad \iota \notin \chi}{(\mathsf{new\ C}\ ,\ \pi), \chi \rightsquigarrow (\iota, \pi), \chi[\iota \mapsto (\mathsf{C}, \overline{\mathsf{f} : \mathsf{null}})]}$$

Fld

$$\frac{\chi(\iota) = (\mathsf{C}, \overline{\mathsf{f} : \mathsf{v}})}{(\iota.\mathsf{f_i},\ \pi), \chi \rightsquigarrow (\mathsf{v_i}, \pi), \chi}$$

Join

$$\frac{\chi(\kappa) \downarrow_3 = \epsilon \quad \chi' = \chi[\ (\kappa, \mathsf{r}) + = \pi]}{(\kappa.\mathsf{r.join}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi), \chi'}$$

FldAss

$$\frac{\chi' = \chi[\iota \mapsto \chi(\iota)[\mathsf{f} \mapsto \mathsf{v}]]}{(\iota.\mathsf{f} := \mathsf{v}\ ,\ \pi), \chi \rightsquigarrow (\mathsf{v}, \pi), \chi'}$$

Meth

$$\frac{\chi(\iota) = (\mathsf{C}, \overline{\mathsf{f} : \mathsf{v}}) \quad \mathsf{mBody}(\mathsf{Prog}, \mathsf{C}, \mathsf{m}) = \mathsf{e}}{(\iota.\mathsf{m}(\overline{\mathsf{v}})\ ,\ \pi), \chi \rightsquigarrow (\mathsf{e}[\iota/\mathsf{this}][\overline{\mathsf{v}}/\overline{\mathsf{x}}]\ ,\ \pi), \chi}$$

NewCV

$$\frac{\mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) = \overline{\mathsf{r}} \quad \kappa \notin \chi \quad \chi' = \chi[\kappa \mapsto (\mathsf{CV}, \overline{\mathsf{r} : \epsilon}, \epsilon)]}{(\mathsf{new}\ \ \mathsf{CV}, \pi)\ , \chi \rightsquigarrow\ \ (\kappa, \pi), \chi'}$$

Leave

$$\frac{\pi \in \chi(\kappa) \downarrow_2 (\mathsf{r}) \quad \chi' = \chi[\ (\kappa, \mathsf{r}) - = \pi]}{(\kappa.\mathsf{r}.\mathsf{leave}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi), \chi'}$$

Start

$$\chi(\kappa) = (\mathsf{CV},\ \overline{\mathsf{r} : \overline{\pi}},\ \epsilon)$$
$$\chi'' = \chi[\kappa \mapsto (\mathsf{CV},\ \overline{\mathsf{r} : \overline{\pi}},\ \mathsf{ChInit})]$$
$$\forall\ \mathsf{ch} \in \mathsf{ChanIDs}(\mathsf{Prog}, \mathsf{CV}) :\ \ \mathsf{ChInit}(\mathsf{ch}, 0) = \epsilon$$
$$\forall\ \mathsf{i} \neq 0 :\ \mathsf{ChInit}(\mathsf{ch}, \mathsf{i}) = \bot$$

$$\forall\ \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) :\ \forall\ \pi \in \mathsf{parts}(\mathsf{Prog}, \chi, \kappa, \mathsf{r}) :\ \forall\ \mathsf{ch} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) :$$
$$\chi'(\pi, \kappa, \mathsf{ch}) = (0, 0, 0)$$
$$\chi'(\mathsf{z}) = \chi''(\mathsf{z}) \quad \text{if } \mathsf{z} \notin (\pi, \kappa, \mathsf{ch})$$

$$(\kappa.\mathsf{start}(), \pi), \chi \rightsquigarrow (\mathsf{null}, \pi),\ \chi'$$

Send

$$\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k})$$
$$\chi_1 \ =\ \chi[(\pi, \kappa, \mathsf{ch}) \ \mapsto\ (\mathsf{i} + 1, \mathsf{j}, \mathsf{k})]$$
$$\chi_2 \ =\ \chi_1(\kappa) \downarrow_3 [(\mathsf{ch}, \mathsf{i}) \mapsto \chi_1(\kappa) \downarrow_3 (\mathsf{ch}, \mathsf{i}) + +[\mathsf{v}]]$$

$$(\kappa.\mathsf{ch.send}(\mathsf{v}),\ \pi),\ \chi \rightsquigarrow (\mathsf{null}, \pi),\ \chi_2$$

Receive1

$$\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k})$$
$$\mathsf{k} = \sharp\ \mathsf{senders}(\mathsf{Prog}, \chi, \kappa,\ \mathsf{ch}) - 1$$
$$\chi' \ =\ \chi[(\pi, \kappa, \mathsf{ch}) \ \mapsto\ (\mathsf{i}, \mathsf{j} + 1, 0)]$$

$$(\mathsf{x} = \kappa.\mathsf{ch.receive}()\ \mathsf{in\ e},\ \pi), \chi \rightsquigarrow (\mathsf{EOM}, \pi), \chi'$$

Receive2

$$\chi(\pi, \kappa, \mathsf{ch}) = (\mathsf{i}, \mathsf{j}, \mathsf{k})$$
$$\mathsf{k} < \sharp\ \mathsf{senders}(\mathsf{Prog}, \chi, \kappa,\ \mathsf{ch}) - 1$$
$$\chi(\kappa) \downarrow_3 (\mathsf{dual}(\mathsf{ch}), \mathsf{j})[\mathsf{k}] = \mathsf{v}$$
$$\chi' \ =\ \chi[(\pi, \kappa, \mathsf{ch}) \mapsto (\mathsf{i}, \mathsf{j}, \mathsf{k} + 1)]$$

$$(\mathsf{x} = \kappa.\mathsf{ch.receive}()\ \mathsf{in\ e},\ \pi), \chi \rightsquigarrow (\ (\mathsf{e}[\mathsf{v}/\mathsf{x}];\ \mathsf{x}\ =\ \kappa.\mathsf{ch.receive}()\ \mathsf{in\ e})\ , \pi), \chi'$$

Appendix C

Static Typing Rules of *Roles*

Axiom

$$\frac{}{\Gamma;\Sigma \vdash \mathsf{this} : \Gamma(\mathsf{this});\Sigma} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{x} : \Gamma(\mathsf{x});\Sigma}$$

Null

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}} \quad \mathsf{Prog} \vdash \mathsf{CV}\diamond_{\mathsf{cv}}}{\Gamma;\Sigma \vdash \mathsf{null} : \mathsf{C};\Sigma} \qquad \frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}} \quad \mathsf{Prog} \vdash \mathsf{CV}\diamond_{\mathsf{cv}}}{\Gamma;\Sigma \vdash \mathsf{null} : \mathsf{CV};\Sigma}$$

Subsumption

$$\frac{\mathsf{Prog} \vdash \mathsf{C} \leq \mathsf{C}' \quad \Gamma;\Sigma \vdash \mathsf{e} : \mathsf{C};\Sigma'}{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{C}';\Sigma'}$$

NewC

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\Gamma;\Sigma \vdash \mathsf{new\ C} : \mathsf{C};\Sigma}$$

Seq

$$\frac{\Gamma;\Sigma \vdash \mathsf{e}_1 : \mathsf{t}_1;\Sigma'' \quad \Gamma;\Sigma'' \vdash \mathsf{e}_2 : \mathsf{t}_2;\Sigma'}{\Gamma;\Sigma \vdash \mathsf{e}_1;\mathsf{e}_2 : \mathsf{t}_2;\Sigma'}$$

Fld

$$\frac{\Gamma;\Sigma \vdash \mathsf{e} : \mathsf{C},\Sigma'}{\Gamma;\Sigma \vdash \mathsf{e.f} : \mathsf{F}(\mathsf{Prog},\mathsf{C},\mathsf{f});\Sigma'}$$

FldAssign

$$\frac{\Gamma;\Sigma \vdash e : C;\Sigma'' \qquad \Gamma;\Sigma'' \vdash e' : F(Prog,C,f);\Sigma'}{\Gamma;\Sigma \vdash e.f = e' : F(Prog,C,f);\Sigma'}$$

Method

$$\frac{\begin{array}{l}\Gamma;\Sigma \vdash e : C;\Sigma_0 \\ \Gamma;\Sigma_{i-1} \vdash e_i : t_i;\Sigma_i \qquad i \in \{1...n\} \\ mType(Prog,C,m) = t_1...t_n \rightarrow t,\Sigma' \\ \Sigma' \preceq_{pre} \Sigma_n\end{array}}{\Gamma;\Sigma \vdash e.m(e_1...e_n) : t;\Sigma_n - \Sigma'}$$

NewCV

$$\frac{Prog \vdash CV\diamond_{cv}}{\Gamma;\Sigma \vdash new\ CV() : CV;\Sigma}$$

Spawn

$$\frac{\Gamma;\emptyset \vdash e : t;\emptyset}{\Gamma;\Sigma \vdash Spawn\{e\} : t;\Sigma}$$

Start

$$\frac{\begin{array}{l}\Gamma(cv) = CV \\ \forall ch \in CH(Prog,CV) :\ session(Prog,CV,ch) = s \quad \rightarrow \quad \Sigma(cv,ch) = s\end{array}}{\Gamma;\Sigma \vdash cv.start() : CV;\Sigma}$$

Leave

$$\frac{\begin{array}{l}\Gamma(cv) = CV \\ r \in RS(Prog,CV) \qquad \forall ch\ \in role\_channels(Prog,CV,r)\ :\ \Sigma_0(cv,ch) \neq Udf \\ \forall ch_i \in role\_channels(Prog,CV,r)\ :\ \Sigma_i = \Sigma_{i-1}[(cv,ch) \mapsto \epsilon] \\ n\ =\ \sharp role\_channels(Prog,CV,r)\end{array}}{\Gamma;\Sigma_0 \vdash cv.r.leave() : CV;\Sigma_n}$$

Join

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \\ \mathsf{r} \in \mathsf{RS}(\mathsf{Prog}, \mathsf{CV}) \qquad \forall \mathsf{ch} \ \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) \ : \ \Sigma_0(\mathsf{cv}, \mathsf{ch}) = Udf \\ \forall \mathsf{ch_i} \in \mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r}) \ : \ \Sigma_\mathsf{i} = \Sigma_{\mathsf{i}-1}[(\mathsf{cv}, \mathsf{ch_i}) \mapsto \mathsf{session}(\mathsf{Prog}, \mathsf{CV}, \mathsf{ch_i})] \\ \mathsf{n} \ = \ \sharp\mathsf{role\_channels}(\mathsf{Prog}, \mathsf{CV}, \mathsf{r})\end{array}}{\Gamma; \Sigma_0 \vdash \mathsf{cv.r.join}() : \mathsf{CV}; \Sigma_\mathsf{n}}$$

Send

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \qquad \Gamma(\mathsf{v}) = \mathsf{C} \\ \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog}, \mathsf{CV}) \\ \Sigma(\mathsf{cv}, \mathsf{ch}) = !\mathsf{C.s}\end{array}}{\Gamma; \Sigma \vdash \mathsf{cv.ch.send(v)} : \mathsf{C}; \ \Sigma[(\mathsf{cv}, \mathsf{ch}) \mapsto \mathsf{s}]}$$

Receive

$$\frac{\begin{array}{l}\Gamma(\mathsf{cv}) = \mathsf{CV} \\ \mathsf{ch} \in \mathsf{CHS}(\mathsf{Prog}, \mathsf{CV}) \\ \Gamma; \Sigma \vdash \mathsf{e} : \mathsf{t}; \Sigma' \\ \Gamma(\mathsf{x}) = \mathsf{C} \qquad \Sigma(\mathsf{cv}, \mathsf{ch}) = \Sigma'(\mathsf{cv}, \mathsf{ch}) = ?\mathsf{C.s}\end{array}}{\Gamma; \Sigma \vdash (\mathsf{x} = \mathsf{cv.ch.receive}() \ \mathsf{in} \ \mathsf{e}) : \mathsf{t}; \ \Sigma'[(\mathsf{cv}, \mathsf{ch}) \mapsto \mathsf{s}]}$$

Appendix D

Runtime Typing Rules of *Roles*

Null

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}} \quad \mathsf{Prog} \vdash \mathsf{CV}\diamond_{\mathsf{cv}}}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{null}, \pi) : \mathsf{C}; \Sigma \quad \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{null}, \pi) : \mathsf{CV}; \Sigma}$$

EOM

$$\frac{\mathsf{Prog} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{EOM}, \pi) : \mathsf{C}; \Sigma}$$

ObjAddr

$$\frac{\chi(\iota) \downarrow_1 = \mathsf{C}}{\chi; \Sigma \vdash_{\mathsf{r}} (\iota, \pi) : \mathsf{C}; \Sigma}$$

ConvAddr

$$\frac{\chi(\kappa) \downarrow_1 = \mathsf{CV}}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa, \pi) : \mathsf{CV}; \Sigma}$$

Subsumption

$$\frac{\mathsf{Prog} \vdash \mathsf{C} \leq \mathsf{C}' \quad \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{C}; \Sigma'}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{C}'; \Sigma'}$$

FldAssign

$$\frac{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}, \pi) : \mathsf{C}; \Sigma \quad \chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e}', \pi) : \mathsf{F}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}); \Sigma'}{\chi; \Sigma \vdash_{\mathsf{r}} (\mathsf{e.f} := \mathsf{e}',\ \pi) : \mathsf{F}(\mathsf{Prog}, \mathsf{C}, \mathsf{f}); \Sigma'}$$

Seq

$$\frac{\chi; \Sigma \vdash_r (e_1, \pi) : t_1; \Sigma'' \quad \chi; \Sigma'' \vdash_r (e_2, \pi) : t_2; \Sigma'}{\chi; \Sigma \vdash_r (e_1; e_2, \pi) : t_2; \Sigma'}$$

Fld

$$\frac{\chi; \Sigma \vdash_r (e, \pi) : C; \Sigma}{\chi; \Sigma \vdash_r (e.f, \pi) : F(Prog, C, f); \Sigma}$$

NewC

$$\frac{Prog \vdash C\diamond_c}{\chi; \Sigma \vdash_r (\text{new } C, \pi) : C; \Sigma}$$

NewCV

$$\frac{Prog \vdash CV\diamond_{cv}}{\chi; \Sigma \vdash_r (\text{new } CV(), \pi) : CV; \Sigma}$$

Method

$$\frac{\chi; \Sigma_0 \vdash_r (e, \pi) : C; \Sigma_0 \quad mType(Prog, C, m) = t_1 ... t_n \to t, \Sigma' \quad \chi; \Sigma_{i-1} \vdash_r (e_i, \pi) : t_i'; \Sigma_i \quad Prog \vdash t_i' \leq t_i \quad \text{for } i \in \{1...n\} \quad \Sigma' \preceq_{pre} \Sigma_n}{\chi; \Sigma_0 \vdash_r (e.m(e_1...e_n), \pi) : t;\ \Sigma_n - \Sigma'}$$

Spawn

$$\frac{\chi; \emptyset \vdash_r (e, \pi') : t; \emptyset \quad \text{for some } \pi' \notin \chi}{\chi; \Sigma \vdash_r (Spawn\{e\}, \pi) : t; \Sigma}$$

Join

$$\frac{\chi(\kappa) \downarrow_1 = CV \quad \pi \notin \chi(\kappa) \downarrow_2 (r) \quad \chi(\kappa) \downarrow_3 = \epsilon \quad \forall\, i_{\{1 \leq i \leq \sharp role\_channels(Prog, CV, r) = n\}} :\ \Sigma_i = \Sigma_{i-1}[(\kappa, ch_i) \mapsto session(Prog, CV, ch_i)]}{\chi; \Sigma_0 \vdash_r (\kappa.r.join(), \pi) : CV; \Sigma_n}$$

Leave

$$\frac{\chi(\kappa)\downarrow_1 = \mathsf{CV} \qquad \pi \in \chi(\kappa)\downarrow_2(\mathsf{r}) \qquad \forall\, \mathsf{i}_{\{1\le \mathsf{i}\le \sharp \mathsf{role\_channels(Prog,CV,r)}\,=\,\mathsf{n}\}} :\ \Sigma_{\mathsf{i}} = \Sigma_{\mathsf{i}-1}[(\kappa, \mathsf{ch_i}) \mapsto \epsilon]}{\chi; \Sigma_0 \vdash_{\mathsf{r}} (\kappa.\mathsf{r.leave}(), \pi) : \mathsf{CV}; \Sigma_{\mathsf{n}}}$$

Start

$$\frac{\chi(\kappa)\downarrow_1 = \mathsf{CV}}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa.\mathsf{start}(), \pi) : \mathsf{CV}; \Sigma}$$

Send

$$\frac{\chi(\kappa)\downarrow_1 = \mathsf{CV} \qquad \mathsf{ch} \in \mathsf{CHS(Prog, CV)} \qquad \Sigma(\kappa, \mathsf{ch}) = !\mathsf{C.s} \qquad \chi; \emptyset \vdash_{\mathsf{r}} \mathsf{v} : \mathsf{C}'; \emptyset \qquad \mathsf{Prog} \vdash \mathsf{C}' \le \mathsf{C} \qquad \Sigma' = \Sigma[(\kappa, \mathsf{ch}) \mapsto \mathsf{s}]}{\chi; \Sigma \vdash_{\mathsf{r}} (\kappa.\mathsf{ch.send(v)}, \pi) : \mathsf{C}; \Sigma'}$$

Receive

$$\frac{\chi(\kappa)\downarrow_1 = \mathsf{CV} \qquad \mathsf{ch} \in \mathsf{CHS(Prog, CV)} \qquad \chi; \Sigma \vdash (\mathsf{e}, \pi) : \mathsf{t}; \Sigma' \qquad \Sigma(\kappa, \mathsf{ch}) = \Sigma'(\mathsf{cv}, \mathsf{ch}) = ?\mathsf{C.s}}{\chi; \Sigma \vdash_{\mathsf{r}} (\ (\mathsf{let\ x} = \kappa.\mathsf{ch.receive()\ in\ e})\ , \pi) : \mathsf{t}; \Sigma'[(\mathsf{cv}, \mathsf{ch}) \mapsto \epsilon]}$$