Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Global types for asynchronous networks: a coinductive approach

Riccardo Bianchini

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

Global types for asynchronous networks: a coinductive approach

Riccardo Bianchini

Advisors: Elena Zucca, Francesco Dagnino
External advisor: Paola Giannini (Università del Piemonte Orientale)

Examiners: Davide Ancona, Eugenio Moggi

October, 2020

Abstract

Global types are a simple but expressive type formalism, used to model the intended interaction structure among multiple participants exchanging messages in a network. Traditionally, global types have been used to ensure safety properties of interactions, such as absence of communication errors, deadlock freedom and race freedom. Using the notion of projection it is possible to obtain local types for the participants, which represent their expected behaviour. In this way, it is possible to check that the network will evolve as defined in the global type.

In the thesis, we have developed an implementation in co-logic programming of a novel formulation of global types for asynchronous networks, where asynchrony is expressed at the level of the type system. Besides providing an executable version of typechecking, the benefit of this encoding is that it forces to clearly understand and express the either inductive or coinductive nature of definitions, and the related termination issues.

Table of Contents

Introduction

The thesis is about the description of asynchronous networks of participants that communicate receiving and sending messages. In particular, the focus is on the use of type formalisms to guarantee interesting properties of execution, for instance:

- Messages will never have a different type than expected (communication safety).
- All sent messages will be eventually read and every process waiting for a message will eventually receive one (progress).
- All performed interactions will be complying with the global type describing the global protocol (protocol fidelity).

The type formalism discussed in the thesis is called in literature *global type*. This model describes the network at three levels:

- protocol description
- session description
- process description

The first level is a high-level description of the whole network; the second level describes a portion of communicating participants; the third level describes the behaviour of a single participant as a process. The whole model is based on the notion of *session*, firstly introduced in [HVK98], that is, a series of message exchanges among some participants. The other core concept of the model is *projection*. Projection takes in input a global type and a participant and returns as output the corresponding session type. This type is used as type for processes; if a process is well-typed with respect to the session type obtained by projection, then the process behaviour is compliant with the global protocol.

In the thesis, we have developed an implementation in *co-logic programming* of a novel formulation of global types for asynchronous netwoks, where asynchrony is expressed at

the level of the type system. Co-logic programming is an extension of logic-programming where predicates can be marked as coinductive. Besides, of course, providing an executable version of typechecking, the benefit of this encoding is that it forces to clearly understand and express the either inductive or coinductive nature of definitions, and the related termination issues.

Chapter 1 is a presentation of the formalism of global types in its classical version in the literature; in particular, we describe the process language, the global and the session types, the reduction rules, that are the execution rules of the network, and the typing rules.

In Chapter 2 we introduce the fundamental ingredients of the coinductive approach used in the thesis work. First, we recall the notion of coinductive definition, as opposed to inductive definition, in the framework of *inference systems*. Then, we recall classic notions and definitions of logic programming, and how the classic paradigm can be extended to support non-well founded structures, such as infinite lists or graphs, and coinductive predicates.

Chapter 3 is a presentation of the new proposal of global types. This proposal has two main differences with respect to the classical approach described in Chapter 1:

- Processes and types are defined coinductively, and, correspondingly, functions handling them, e.g., projection is defined coinductively. The coinductive approach allows a natural way to define infinite terms, e.g., a process term representing a server running forever, replacing the explicit fixed-point operator used in the classical approach. Moreover, as said above, the purpose of these formalisms is to guarantee properties, and to guarantee properties on infinite objects it is often required to reason on infinite proof trees, handled in a natural way by coinductive techniques. Finally, such coinductive approach will be exploited in Chapter 4 to provide a implementation in co-logic programming.
- Global types specify send and receive operations separately. In this way, they are less abstract then the global types in the first chapter, that is, the notion of asynchrony is present also in the protocol description. Instead, usual global types, when used to describe asynchronous networks, have to be combined with a subtyping relation on session types, which however has been shown to be undecidable. One aim of the novel proposal is indeed to provide a decidable approach.

Chapter 4 is the core of the thesis, and consists in a detailed description of the SWI-Prolog implementation of definitions in Chapter 3. The explanation focuses on termination issues; notably, sometimes inductive predicate are adequate, whereas in many cases it is necessary to use the coinductive extension of SWI-Prolog, relying on a mechanism of cycle detection which ensures (successful) termination when the same goal is encountered twice. Moreover, some predicates need to be implemented, rather than directly, as the negation of a predicate defined coinductively. Finally, in some cases, it is necessary to implement a by-hand

cycle detection mechanism, since neither a standard inductive definition, nor a coinductive definition using the built-in cycle detection are enough to ensure termination. For instance, projection is implemented by an inductive predicate using an ad-hoc mechanism, which, to detect a cycle, considers, rather than the whole goal, only some of its arguments. The complete code, together with a test suite, and instructions for using the prototype, can be found at `https://github.com/RiccardoBianc/Asynchronous-global-types-implementation`.

Finally, in Chapter 5 we summarize the contribution of the thesis and discussed future developments.

Chapter 1

Introduction to global types

In this chapter we report the main notions and formal definitions about global types as they are given in the literature. In particular, our presentation is based on the papers [CDPY15], [HYC08] and [BCD+08]. In Sect. 1.1 we describe the programming paradigm based on communication, using the "Two Buyers" example from [HVK98]. In Sect. 1.2 we define the process calculus with its reduction and congruence rules, in Sect. 1.3 we provide the syntax of both global and session types, and the definition of projection, and in Sect. 1.4 the typing rules and an informal presentation of the main results.

1.1 Communication-based programming

Nowadays, programming distributed software is increasingly common, so it is important to have languages and formalisms appropriate to these contexts. Many such languages and formalisms have been proposed so far for the description of software based on communication.

This programming paradigm is based on the concept of *session*, introduced in [HVK98]. A session is a series of interactions among some participants. If the participants are two, then the session is called *binary* or *dyadic*, otherwise the session is called *multiparty*. The communication is allowed by one or more *channels*, that can be shared among participants or local to participants. In the calculus they are implemented with queues. A session is accessed with a shared name, used to identify the session itself. After an acceptance phase, the channels are initialised as empty and the session starts.

This model is structured into three levels of abstraction, each one with its formalism:

- *Global type* level

This level is the description of the whole network, from an external point of view; all interactions are described with the participants involved and the messages exchanged.

- *Session type* level
  This level describes the behaviour of a single session, among one or more participants. This is a local description of the network obtained after the projection of a global type on a single participant.

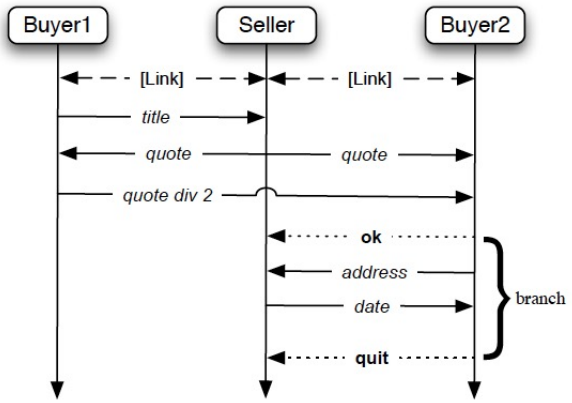
- *Process* level
  This level describes the behaviour of a single participant. Very often the formalism used to describe a process is a dialect of the π-calculus, see [Mil99].

In order to present the calculus, the session types and the global types, we will use the example of the "Two Buyers" protocol from [HYC08].

This example system is composed of three participants: Buyer1, Buyer2 and Seller.

- In the first phase the three participants establish a session, then
- Buyer1 sends a message containing the title of a book to Seller,
- Seller sends to both the buyers their quote of the price,
- Buyer1 sends to Buyer2 its quote of the price;
- after receiving its quote, Buyer2 has two choices:
  - either to send `ok` and the address to Seller, and then waiting for a message from Seller containing the date,
  - or to send `quit` and to give up.

This protocol can be described by the following sequence diagram:



$\mathsf{P}, \mathsf{Q}, \mathsf{R}$	$::=$	$\overline{a}[\mathsf{p}](y).\mathsf{P}$	(Session request)	$\mathcal{M} ::= \emptyset \mid \mathcal{M} \cdot m$	(Queue)
		$\mid a[\mathsf{p}](y).\mathsf{P}$	(Session acceptance)	$m ::= \langle \mathsf{p}, v, \mathsf{q} \rangle$	(Message)
		$\mid c!\langle \mathsf{p}, e \rangle.\mathsf{P}$	(Value sending)	$\mid \langle \mathsf{p}, l, \mathsf{q} \rangle$	
		$\mid c?(\mathsf{p}, x).\mathsf{P}$	(Value reception)	$\mid \langle \mathsf{p}, s[\mathsf{p}'], \mathsf{q} \rangle$	
		$\mid c!\langle\langle \mathsf{p}, c' \rangle\rangle.\mathsf{P}$	(Channel delegation)	p, q	(Partecipant number)
		$\mid c?((\mathsf{q}, y)).\mathsf{P}$	(Channel reception)	$c ::= y \mid s[\mathsf{p}]$	(Channel)
		$\mid c \oplus \langle \mathsf{p}, l \rangle.\mathsf{P}$	(Label sending)	$v ::= \texttt{true} \mid \texttt{false}$	(Value)
		$\mid c\&(\mathsf{p}, \{l_i : \mathsf{P}\}_{i \in I})$	(Branch)	x	(Value variable)
		$\mid \mathsf{if}\ e\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}$	(Conditional)	e	(Expression)
		$\mid \mathsf{P} \parallel \mathsf{Q}$	(Parallel)	$s[\mathsf{p}]$	(Channel (with role))
		$\mid \mathbf{0}$	(Zero)	X, Y	(Process variable)
		$\mid \mu \mathsf{X}.\mathsf{P}$	(Recursion)	l	(Label)
		$\mid \mathsf{X}$	(Process variable)	s	(Session name)
		$\mid (\nu s)(\mathsf{P})$	(Session hiding)	a	(Service name)
		$\mid s : \mathcal{M}$	(Session queue)	y, z, t	(Channel variable)

The grey background denotes runtime syntax, not occurring in code written by the programmer.

Figure 1.1: Syntax of processes

This example is interesting, because it is not easy to describe the communications as exchanges of messages between pairs of participants. Instead, it is necessary to use a single multiparty session to describe the entire protocol. Note that communications are always between different participants, i.e., a participant cannot send/receive a message to/from himself.

1.2 Process calculus

The behaviour of this example can be formalized using a process language. In particular, each participant can be described in isolation, and then the three behaviours can be executed in parallel. We will use a restriction with only global service names of the calculus in [CDPY15] (originally introduced in [BCD$^+$08]).

The syntax of the process calculus is given in Fig. 1.1. A channel variable y is a binder in session request, session acceptance and channel reception. A value variable x is a binder in value reception. A process variable X is a binder in recursion. Finally, a session name is

a binder in session hiding. We omit the standard definition of free variables and names.

Processes of shape $a[\mathsf{p}](y).\mathsf{P}$ and $\overline{a}[\mathsf{p}](y).\mathsf{P}$ cooperate in starting a multiparty session, using a service name identified by a. Participants are represented by progressive numbers, ranged over by p, q. Among the processes that share the same service name, the one with the highest number is indicated with the syntax $\overline{a}[\mathsf{p}](y).\mathsf{P}$. In this way, during the request phase it is possible to know the number of participants needed to start the session looking only at that process.

Once a session is established, the channel variable y in $a[\mathsf{p}](y).\mathsf{P}$ is replaced, in P, by a channel (with role), of shape $s[\mathsf{p}]$; this syntax represents the channel of the participant p in the session s.

Sessions have three primitives of communication, corresponding to three types of messages:

- sending and receiving a value
- delegating and receiving a channel
- sending a label and receiving one of the labels in a set (branch)

We formalize asynchronous communications, so sends are non-blocking. Channel delegation is a mechanism allowing the receiving process to participate in a session it was not part of. In this way, it is possible for the receiving participant to communicate as if it was the sender, allowing to distribute the session among the participants in a way that can be disciplined by the programmer. In selection and branching, a process sends one of the labels expected by the receiving process.

In communication operations, c can be either a channel variable y or, during the execution, a channel $s[\mathsf{p}]$. Each participant in a session has its own channel, that is assigned at the beginning of the session and from which it reads the messages sent to him by the other participants. If two participants could share the same channel, as in the calculus of [HYC08] where explicit channels are used, then the reading/writing order of the messages could depend on the evaluation order, leading to non-deterministic results. To avoid this, in [HYC08] some linearity conditions were introduced (Definition 3.5).

For the conditional construct we assume to have an expression language, with, at least, boolean expressions. Processes may be parallel compositions and the process $\mathbf{0}$ is the inactive process. Recursion is assumed to be *guarded*, in the sense that a unique process should be defined; for instance, the process term as $\mu\mathsf{X}.\mathsf{X}$ is ill-formed.

The syntax $(\nu s)(\mathsf{P})$ makes the session name s local to P and can only be used in runtime expressions.

After the acceptance phase, when a session is established, a corresponding queue is created. Queues are part of the runtime language, that is, they cannot be handled by the

programmer. Messages in the queue are triples consisting of the sender p, the receiver q, and either a value, or a label, or a channel.

A value message $\langle \mathsf{p}, v, \mathsf{q} \rangle$ indicates that the value v was sent by the participant p and the recipient is the participant q. A channel message (delegation) $\langle \mathsf{p}, s[\mathsf{p}'], \mathsf{q} \rangle$ indicates that p delegates to q the role of p' in the session s. A label message $\langle \mathsf{p}, l, \mathsf{q} \rangle$ is similar to a value message, however labels are only used to conditionally execute code, and contain no information, whereas values have a type as in programming languages.

The empty queue is denoted by $\varnothing$, and $\mathcal{M} \cdot m$ denotes the queue obtained by concatenating m to the queue $\mathcal{M}$.

The participants above can be written using the process calculus in this way:

$$
\begin{array}{lcl}
\mathsf{Buyer1} & = & a[1](y).y!\langle 3, \mathit{title}\rangle.y?(3,x).y!\langle 2, x/2\rangle.\mathbf{0} \\
\mathsf{Buyer2} & = & a[2](y).y?(3,x).y?(1,x).\mathsf{if}\ (x<6)\ \mathsf{then}\ y \oplus \langle 3, \mathbf{ok}\rangle.y!\langle 3, \mathit{address}\rangle.y?(3,x).\mathbf{0}\ \mathsf{else}\ y \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \\
\mathsf{Seller} & = & \overline{a}[3](y).y?(1,x).y!\langle 2, 10\rangle.y!\langle 1, 10\rangle.y\&(2, \{\mathbf{ok} : y?(3,x).y!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\})
\end{array}
$$

The initial network of the example can be written in this way:

$$\mathsf{Buyer1} \parallel \mathsf{Buyer2} \parallel \mathsf{Seller}$$

The operational semantics consists of reduction rules and structural equivalence rules that permit rearranging the terms in order to apply a specific reduction rule.

Structural equivalence is denoted by $\equiv$ and defined adding α-conversion to the rules below, where $\mathrm{fn}(\mathsf{Q})$ is the set of free session names in Q.

$$
\begin{array}{c}
(\mathsf{P} \parallel \mathsf{Q}) \parallel \mathsf{R} \equiv \mathsf{P} \parallel (\mathsf{Q} \parallel \mathsf{R}) \qquad \mathsf{P} \parallel \mathsf{Q} \equiv \mathsf{Q} \parallel \mathsf{P} \qquad \mathsf{P} \parallel \mathbf{0} \equiv \mathsf{P} \\
(\nu s)(\mathsf{P}) \parallel \mathsf{Q} \equiv (\nu s)(\mathsf{P} \parallel \mathsf{Q}) \qquad \text{if } s \notin \mathrm{fn}(\mathsf{Q}) \\
(\nu s)((\nu s')(\mathsf{P})) \equiv (\nu s')((\nu s)(\mathsf{P})) \qquad (\nu s)(\mathbf{0}) \equiv \mathbf{0} \qquad (\nu s)(s : \emptyset) \equiv \mathbf{0} \\
\mu \mathsf{X}.\mathsf{P} \equiv \mathsf{P}[\mu \mathsf{X}.\mathsf{P}/\mathsf{X}]
\end{array}
$$

The congruence rules state that the parallel operator is associative, commutative and the inactive process $\mathbf{0}$ is the neutral element. In session hiding, a binder can be extended to a process in paralleli if this does not capture free names, exchanged with another, and removed if applied to the inactive process or to an empty queue for the corresponding session. Finally, a recursive process is congruent to its unfolding.

The order of elements in the queue can be changed following this equivalence rule:

(Init) $a[1](y).\mathsf{P}_1 \parallel \ldots \parallel a[n-1](y).\mathsf{P}_{n-1} \parallel \overline{a}[n](y).\mathsf{P}_n \longrightarrow$
$(\nu s)(\mathsf{P}_1\{s[1]/y\} \parallel \ldots \parallel \mathsf{P}_{n-1}\{s[n-1]/y\} \parallel \mathsf{P}_n\{s[n]/y\} \parallel s : \emptyset) \quad (s \text{ fresh})$

(Snd) $s[\mathsf{p}]!\langle \mathsf{q}, e\rangle.\mathsf{P} \parallel s : \mathcal{M} \longrightarrow \mathsf{P} \parallel s : \mathcal{M} \cdot \langle \mathsf{p}, v, \mathsf{q}\rangle \qquad (e \downarrow v)$

(Rcv) $s[\mathsf{p}]?(\mathsf{q}, x).\mathsf{P} \parallel s : \langle \mathsf{q}, v, \mathsf{p}\rangle \cdot \mathcal{M} \longrightarrow \mathsf{P}[v/x] \parallel s : \mathcal{M}$

(Deleg) $s[\mathsf{p}]!\langle\langle \mathsf{q}, s'[\mathsf{p}']\rangle\rangle.\mathsf{P} \parallel s : \mathcal{M} \longrightarrow \mathsf{P} \parallel s : \mathcal{M} \cdot \langle \mathsf{p}, s'[\mathsf{p}'], \mathsf{q}\rangle$

(RcvCh) $s[\mathsf{p}]?((\mathsf{q}, y)).\mathsf{P} \parallel s : \langle \mathsf{q}, s'[\mathsf{p}'], \mathsf{p}\rangle \cdot \mathcal{M} \longrightarrow \mathsf{P}\{s'[\mathsf{p}']/y\} \parallel s : \mathcal{M}$

(SndL) $s[\mathsf{p}] \oplus \langle \mathsf{q}, l\rangle.\mathsf{P} \parallel s : \mathcal{M} \longrightarrow \mathsf{P} \parallel s : \mathcal{M} \cdot \langle \mathsf{p}, l, \mathsf{q}\rangle$

(Branch) $s[\mathsf{p}]\&(\mathsf{q}, \{l_i : \mathsf{P}_i\}_{i\in I}) \parallel s : \langle \mathsf{q}, l_j, \mathsf{p}\rangle \cdot \mathcal{M} \longrightarrow \mathsf{P}_j \parallel s : \mathcal{M} \quad (j \in I)$

(If-T) $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q} \longrightarrow \mathsf{P} \quad (e \downarrow \mathtt{true})$

(If-F) $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q} \longrightarrow \mathsf{Q} \quad (e \downarrow \mathtt{false})$

(Par) $\dfrac{\mathsf{P} \longrightarrow \mathsf{P}'}{\mathsf{P} \parallel \mathsf{Q} \longrightarrow \mathsf{P}' \parallel \mathsf{Q}}$ (Hide) $\dfrac{\mathsf{P} \longrightarrow \mathsf{P}'}{(\nu s)(\mathsf{P}) \longrightarrow (\nu s)(\mathsf{P}')}$ (Congr) $\dfrac{\mathsf{P} \equiv \mathsf{P}' \quad \mathsf{P}' \longrightarrow \mathsf{Q}' \quad \mathsf{Q}' \equiv \mathsf{Q}}{\mathsf{P} \longrightarrow \mathsf{Q}}$

Figure 1.2: Reduction rules

$$\mathcal{M} \cdot \langle \mathsf{p}, \zeta, \mathsf{q}\rangle \cdot \langle \mathsf{p}', \zeta', \mathsf{q}'\rangle \cdot \mathcal{M}' \equiv \mathcal{M} \cdot \langle \mathsf{p}', \zeta', \mathsf{q}'\rangle \cdot \langle \mathsf{p}, \zeta, \mathsf{q}\rangle \cdot \mathcal{M}' \text{ if } \mathsf{p} \neq \mathsf{p}' \text{ or } \mathsf{q} \neq \mathsf{q}'$$

In other words, the only order that matters is that between messages with the same sender and receiver. This structural equivalence amounts to say that a message queue can be seen as a map from pairs p, q of participants to sub-queues which are ordered lists of labels.

Reduction rules, modelling asynchronous execution of communications, are given in Fig. 1.2:

In rule (Init), n participants, using the shared service name a, start a new session s, and a new queue is created, which will be used for their interactions in this session. We denote by $\mathsf{P}[s[\mathsf{p}]/y]$ the process obtained from P by replacing the channel variable y with the channel $s[\mathsf{p}]$. In rules (Snd), (Deleg) and(SndL), messages are added to the queue; in the corresponding receiving rules, (Rcv), (RcvCh), and (Branch), messages are removed from the queue when they

are read. The rules (If-T) and (If-F) have the standard interpretation, where $e \downarrow v$ denotes that expression e evaluates to value v.

The reduction of the example is given below.

$a[1](y).y!\langle 3, \mathit{title}\rangle.y?(3,x).y!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$a[2](y).y?(3,x).y?(1,x).\mathsf{if}\ (x<6)\ \mathsf{then}\ y \oplus \langle 3, \mathbf{ok}\rangle.y!\langle 3, \mathit{address}\rangle.y?(3,x).\mathbf{0}\ \mathsf{else}\ y \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$\overline{a}[3](y).y?(1,x).y!\langle 2, 10\rangle.y!\langle 1, 10\rangle.y\&(2, \{\mathbf{ok} : y?(3,x).y!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
applying (Init) $\longrightarrow$

$(\nu s)(s[1]!\langle 3, \mathit{title}\rangle.s[1]?(3,x).s[1]!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$s[2]?(3,x).s[2]?(1,x).\mathsf{if}\ (x<6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, \mathit{address}\rangle.s[2]?(3,x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]?(1,x).s[3]!\langle 2, 10\rangle.s[3]!\langle 1, 10\rangle.s[3]\&(2, \{\mathbf{ok} : s[3]?(3,x).s[3]!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \varnothing)$

applying (Send) *to* Buyer1 $\longrightarrow$

$(\nu s)(s[1]?(3,x).s[1]!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$s[2]?(3,x).s[2]?(1,x).\mathsf{if}\ (x<6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, \mathit{address}\rangle.s[2]?(3,x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]?(1,x).s[3]!\langle 2, 10\rangle.s[3]!\langle 1, 10\rangle.s[3]\&(2, \{\mathbf{ok} : s[3]?(3,x).s[3]!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \langle 1, \mathit{title}, 3\rangle)$

applying (Rcv) *to* Seller $\longrightarrow$

$(\nu s)(s[1]?(3,x).s[1]!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$s[2]?(3,x).s[2]?(1,x).\mathsf{if}\ (y<6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, \mathit{address}\rangle.s[2]?(3,x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]!\langle 2, 10\rangle.s[3]!\langle 1, 10\rangle.s[3]\&(2, \{\mathbf{ok} : s[3]?(3,x).s[3]!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \emptyset)$

applying (Send) *to* Seller *two times* $\longrightarrow$

$(\nu s)(s[1]?(3,x).s[1]!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$s[2]?(3,x).s[2]?(1,x).\mathsf{if}\ (y<6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, \mathit{address}\rangle.s[2]?(3,x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]\&(2, \{\mathbf{ok} : s[3]?(3,x).s[3]!\langle 2, \mathit{date}\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \langle 3, 10, 1\rangle \cdot \langle 3, 10, 2\rangle)$

applying (Rcv) *to both* Buyer1 *and* Buyer2 $\longrightarrow$

$(\nu s)(s[1]!\langle 2, x/2\rangle.\mathbf{0} \parallel$
$s[2]?(1, x).\mathsf{if}\ (y < 6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, address\rangle.s[2]?(3, x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]\&(2, \{\mathbf{ok} : s[3]?(3, x).s[3]!\langle 2, date\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \emptyset)$

applying (Send) *to* Buyer1 $\longrightarrow$

$(\nu s)(s[2]?(1, x).\mathsf{if}\ (y < 6)\ \mathsf{then}\ s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, address\rangle.s[2]?(3, x).\mathbf{0}\ \mathsf{else}\ s[2] \oplus \langle 3, \mathtt{quit}\rangle.\mathbf{0} \parallel$
$s[3]\&(2, \{\mathbf{ok} : s[3]?(3, x).s[3]!\langle 2, date\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \langle 1, 5, 2\rangle)$

applying (Rcv) *to* Buyer2 *and evaluating the if guard to* `true`) $\longrightarrow$

$(\nu s)(s[2] \oplus \langle 3, \mathbf{ok}\rangle.s[2]!\langle 3, address\rangle.s[2]?(3, x).\mathbf{0} \parallel$
$s[3]\&(2, \{\mathbf{ok} : s[3]?(3, x).s[3]!\langle 2, date\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \emptyset)$

applying (Sel) *to* Buyer2 $\longrightarrow$

$(\nu s)(s[2]!\langle 3, address\rangle.s[2]?(3, x).\mathbf{0} \parallel$
$s[3]\&(2, \{\mathbf{ok} : s[3]?(3, x).s[3]!\langle 2, date\rangle.\mathbf{0}, \mathtt{quit} : \mathbf{0}\}) \parallel$
$s : \langle 2, \mathtt{ok}, 3\rangle)$

applying (Branch) *to* Seller $\longrightarrow$

$(\nu s)(s[2]!\langle 3, address\rangle.s[2]?(3, x).\mathbf{0} \parallel$
$s[3]?(3, x).s[3]!\langle 2, date\rangle.\mathbf{0} \parallel$
$s : \emptyset)$

applying (Send) *to* Buyer2 *and then* (Rcv) *to* Seller $\longrightarrow$

$(\nu s)(s[2]?(3, x).\mathbf{0} \parallel$
$s[3]!\langle 2, date\rangle.\mathbf{0} \parallel$
$s : \emptyset)$

applying (Send) *to* Seller *and then* (Rcv) *to* Buyer2 *the computation stops*

1.3 Session types and global types

As explained above, in order to guarantee some properties of the network, a type formalism can be used, that describes in an abstract and general way the protocol of the system.

Types used to describe the whole network are known in literature as *global types*, and have been introduced in [HYC08]. Global types describe the interactions among all the participants, listing the participants with their messages. In a sense, they provide a description from an external point a view. Asynchrony in communications is not considered, that is,

the model is abstract from this point of view.

The syntax of global types is given below. Sort types are types of values (e.g., booleans). Exchange types are either sort types or *closed* session types (see below), ranged over by T.

$$
\begin{array}{rcll}
S & ::= & \texttt{bool} \mid \ldots & \text{(Sort)} \\
\mathsf{U} & ::= & S \mid \mathsf{T} & \text{(Exchange type)} \\
\mathsf{G} & ::= & \mathsf{p} \longrightarrow \mathsf{q} : \langle S \rangle.\mathsf{G} & \text{(Value passing)} \\
 & & \mid \mathsf{p} \longrightarrow \mathsf{q} : \langle \mathsf{T} \rangle.\mathsf{G} & \text{(Channel passing)} \\
 & & \mid \mathsf{p} \longrightarrow \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I} & \text{(Branching)} \\
 & & \mid \mu \mathbf{g}.\mathsf{G} \mid \mathbf{g} & \text{(Recursion)} \\
 & & \mid \mathsf{End} & \text{(End)}
\end{array}
$$

The global type $\mathsf{p} \longrightarrow \mathsf{q} : \langle S \rangle.\mathsf{G}$ means that the participant p sends a value of sort S to participant q, and then the interactions described in G take place; analogously, the syntax $\mathsf{p} \longrightarrow \mathsf{q} : \langle \mathsf{T} \rangle.\mathsf{G}$ means that p delegates to q a channel of type T and then the interactions described in G take place. The global type $\mathsf{p} \longrightarrow \mathsf{q} : \{l_i : \mathsf{G}_i\}_{i \in I}$ means that the participant p sends one of the labels l_i to participant q. If l_j is sent, then the interactions described in G_j take place. The global type $\mu \mathbf{g}.\mathsf{G}$ is a recursive type, where, as for processes, recursion is assumed to be *guarded*, in the sense that a unique global type should be defined. We take an *equi-recursive* view of recursive types, that is, we do not distinguish between $\mu \mathbf{g}.\mathsf{G}$ and its unfolding $\mathsf{G}[\mu \mathbf{g}.\mathsf{G}/\mathbf{g}]$ [Pie02, §21.8]. The global type End represents the termination of the session.

The global type of our example, in which we write participants with legible symbols instead of numbers, is the following:

$$
\begin{array}{l}
\mathsf{B1} \to \mathsf{S} : \langle \texttt{string} \rangle. \\
\mathsf{S} \to \mathsf{B2} : \langle \texttt{int} \rangle.\, \mathsf{S} \to \mathsf{B1} : \langle \texttt{int} \rangle. \\
\mathsf{B1} \to \mathsf{B2} : \langle \texttt{int} \rangle. \\
\mathsf{B2} \to \mathsf{S} : \{\texttt{ok} : \mathsf{B2} \to \mathsf{S} : \langle \texttt{string} \rangle.\mathsf{S} \to \mathsf{B2} : \langle \texttt{date} \rangle.\mathsf{End}\ ,\ \texttt{quit} : \mathsf{End}\}
\end{array}
$$

Another type formalism is used to describe the type of a single participant. Such types are

called *session types*, and their syntax is the following:

$$
\begin{array}{rcll}
T & ::= & !\langle \mathsf{p}, S\rangle.T & \text{(Value sending)} \\
& & \mid\ !\langle \mathsf{p}, \mathsf{T}\rangle.T & \text{(Channel delegation)} \\
& & \mid\ ?(\mathsf{p}, \mathsf{U}).T & \text{(Value or channel reception)} \\
& & \mid\ \oplus\langle \mathsf{p}, \{l_i : T_i\}_{i\in I}\rangle & \text{(Label sending (selection))} \\
& & \mid\ \&(\mathsf{p}, \{l_i : T_i\}_{i\in I}) & \text{(Label receiving (branch))} \\
& & \mid\ \mu \mathbf{t}.T \mid \mathbf{t} & \text{(Recursion)} \\
& & \mid\ \mathsf{End} & \text{(End)}
\end{array}
$$

Session types represent the input-output actions performed by single participants. The *send types* express, respectively, the sending of a value of sort S to participant p or the sending of a channel of type T to participant p, followed by the communications described by T. The *selection type* represents the transmission to participant p of a label l_i chosen in the set $\{l_i \mid i \in I\}$, followed by the communications described by T_i. The *reception* and *branching* types are dual of send and selection types. Recursion is guarded also in session types, and we consider them modulo folding/unfolding as done for global types.

1.4 Projection and type checking

The three formalisms introduced so far (processes, global types and session types) are linked together using two operations: *projection* and *process typing*. Projection computes the local type of a single participant starting from a global type. This operation links the global type level with the session type level. Process typing checks that a single process complies with the specification given by a session type, so it links the session type level with the process level.

Projection The *projection* of a global type G on a participant q is the session type $\mathsf{G}\upharpoonright\mathsf{q}$ defined inductively as follows:

$$
(\mathsf{p} \to \mathsf{p}' : \langle U\rangle.\mathsf{G}')\upharpoonright\mathsf{q} = \begin{cases} !\langle \mathsf{p}', U\rangle.(\mathsf{G}'\upharpoonright\mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p} \\ ?(\mathsf{p}, U).(\mathsf{G}'\upharpoonright\mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p}' \\ \mathsf{G}'\upharpoonright\mathsf{q} & \text{otherwise} \end{cases}
$$

$$
(\mathsf{p} \to \mathsf{p}' : \{l_i : \mathsf{G}_i\}_{i\in I})\upharpoonright\mathsf{q} = \begin{cases} \oplus\langle \mathsf{p}', \{l_i : \mathsf{G}_i\upharpoonright\mathsf{q}\}_{i\in I}\rangle & \text{if } \mathsf{q} = \mathsf{p} \\ \&(\mathsf{p}, \{l_i : \mathsf{G}_i\upharpoonright\mathsf{q}\}_{i\in I}) & \text{if } \mathsf{q} = \mathsf{p}' \\ T & \text{if } \mathsf{q} \neq \mathsf{p}, \mathsf{q} \neq \mathsf{p}' \\ & \text{and } \mathsf{G}_i\upharpoonright\mathsf{q} = T \text{ for all } i \in I \end{cases}
$$

$$(\mu \mathbf{g}.\mathsf{G})\upharpoonright \mathsf{q} = \begin{cases} \mu \mathbf{t}_g.(\mathsf{G}\upharpoonright \mathsf{q}) & \text{if } \mathsf{G}\upharpoonright \mathsf{q} \neq \mathbf{t}_g \\ \mathsf{End} & \text{otherwise} \end{cases}$$

$$\mathsf{End}\upharpoonright \mathsf{q} = \mathsf{End}$$
$$\mathbf{g}\upharpoonright \mathsf{q} = \mathbf{t}_g$$

We assume an injective map which associates to each global type variable $\mathbf{g}$ a session type variable $\mathbf{t}_g$.

Note that the projection is not always defined. For example, the projection of a branching on a participant which is neither the sender nor the receiver is defined only if the projections of all branches on the participant are equal. This is due to the fact that this participant is not aware of the label being sent so it cannot behave differently in different branches. A global type G is called *well-formed* if the projection on all its participants is defined.

The projections of the global type G of our example are

$$\mathsf{G}\upharpoonright \mathsf{S} = ?(\mathsf{B1}, \mathtt{string}).!\langle \mathsf{B2}, \mathtt{int}\rangle.!\langle \mathsf{B1}, \mathtt{int}\rangle.\&(\mathsf{B2}, \{\mathtt{ok}; ?(\mathsf{B2}, \mathtt{string}).!\langle \mathsf{B2}, \mathtt{date}\rangle.\mathsf{End}\ , \mathtt{quit}; \mathsf{End}\})$$
$$\mathsf{G}\upharpoonright \mathsf{B1} = !\langle \mathsf{S}, \mathtt{string}\rangle.?(\mathsf{S}, \mathtt{int}).!\langle \mathsf{B2}, \mathtt{int}\rangle.\mathsf{End}$$
$$\mathsf{G}\upharpoonright \mathsf{B}\,2 = ?(\mathsf{S}, \mathtt{int}).?(\mathsf{B1}, \mathtt{int}).\oplus \langle \mathsf{B2}, \{\mathtt{ok}; !\langle \mathsf{S}, \mathtt{string}\rangle.?(\mathsf{S}, \mathtt{date}).\mathsf{End}\ , \mathtt{quit}; \mathsf{End}\}\rangle$$

Type checking For brevity we do not describe type checking of runtime syntax, e.g., queues and processes containing channels with roles. The typing judgements for expressions and processes have the following shape:

$$\Gamma \vdash e : S \quad \text{and} \quad \Gamma \vdash \mathsf{P} \triangleright \Delta$$

where

- Δ is a map from channels to session types
- Γ is a type environment, that is, a map from values to sorts, from service names to global types, and from process variables to maps Δ

Given a type environment, type checking assigns to an expression a sort, and to a process a map assigning session types to the channels possibly used by the process.

Maps are represented by list of pairs as follows:

$$\Gamma ::= \varnothing \mid \Gamma, x : S \mid \Gamma, a : \mathsf{G} \mid \Gamma, \mathsf{X} : \Delta$$
$$\Delta ::= \varnothing \mid \Delta, c : T$$

We assume that the occurrences of x, a, and X are unique in the list, hence Δ, Δ' is well-formed only if the domains of the two maps are disjoint.

The typing rules are given in Fig. 1.4. Recall that T denotes a session type, whereas T denotes a closed session type. The latter notation is used in channel sending and receiving. Rule (NameT) assigns the type to the variables using the map Γ. Rule (RequestT) permits to type the request on a service a of type G, checking that y is the projection of Gon the participant p, that must have the highest role. The rule (AcceptanceT) is the same, but $\mathsf{p}$is not the highest role. The rules (SndT),(RcvT),(DelegT),(ChRcvT), (LSndT),(BranchT) check that the input/output process are compliant with the corresponding type. Note that, in rule (DelegT), the channel c' which is sent is removed from the context in the premise. Indeed, when a channel is delegated, it can no longer be used by the sender, but only by the receiver. Rule rule (ChRcvT) is symmetrical, that is, the received channel is added to the context in the premise, meaning that from now on it can be used by the receiver. The rule (ParT) checks that the two processes in parallel have different channels.

The top-level type environment will contain the association between service names and their global types. The rules (RequestT) and (AcceptanceT) require each participant to use its channels according to the projection of the global type associated with the initiated service. The other rules are straightforward. In rule (ZeroT), Δ End only means that the session types of the channels in Δ must be End.

If the processes in a network are typed by session types which are the projections of a well-formed global type, then the network has the following properties.

- **Communication Safety**: Interactions within sessions never incur a communication error.
- **Progress**: The network is deadlock-free, that is, if there are non-inactive processes the network will evolve.
- **Session Fidelity**: The communication sequences follow the scenario declared by the global type.

Going back to our example, we show a part of the proof that the network is well-typed.

Consider the processes associated to the two buyers and the seller:

$$\begin{array}{lcl} \mathsf{Buyer1} & = & a[1](y).y!\langle 3, title\rangle.y?(3,x).y!\langle 2, x/2\rangle.\mathbf{0} \\ \mathsf{Buyer2} & = & a[2](y).y?(3,x).y?(1,x).P_2 \\ \mathsf{Seller} & = & \overline{a}[3](y).y?(1,x).y!\langle 2, 10\rangle.y!\langle 1, 10\rangle.P_{\mathsf{s}} \end{array}$$

$$\Gamma, a : \mathsf{G} \vdash a : \mathsf{G} \ (\textsc{NameT}) \qquad \Gamma, x : S \vdash x : S \ (\textsc{VarT})$$

$$\Gamma \vdash \mathtt{true} : \mathtt{bool} \ (\textsc{TrueT}) \qquad \Gamma \vdash \mathtt{false} : \mathtt{bool} \ (\textsc{FalseT})$$

$$\frac{\Gamma \vdash a : \mathsf{G} \qquad \Gamma \vdash \mathsf{P} \triangleright \Delta, y : \mathsf{G} \upharpoonright \mathsf{p} \qquad \mathsf{p} = \mathit{max\_part}(\mathsf{G})}{\Gamma \vdash \overline{a}[\mathsf{p}](y).\mathsf{P} \triangleright \Delta} \ (\textsc{RequestT})$$

$$\frac{\Gamma \vdash a : \mathsf{G} \qquad \Gamma \vdash \mathsf{P} \triangleright \Delta, y : \mathsf{G} \upharpoonright \mathsf{p} \qquad \mathsf{p} < \mathit{max\_part}(\mathsf{G})}{\Gamma \vdash a[\mathsf{p}](y).\mathsf{P} \triangleright \Delta} \ (\textsc{AcceptanceT})$$

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash \mathsf{P} \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \mathsf{p}, e\rangle.\mathsf{P} \triangleright \Delta, c : !\langle \mathsf{p}, S\rangle.T} \ (\textsc{SndT}) \qquad \frac{\Gamma, x : S \vdash \mathsf{P} \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathsf{p}, x).\mathsf{P} \triangleright \Delta, c : ?(\mathsf{p}, S).T} \ (\textsc{RcvT})$$

$$\frac{\Gamma \vdash \mathsf{P} \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\langle \mathsf{p}, c'\rangle\rangle.\mathsf{P} \triangleright \Delta, c : !\langle \mathsf{p}, \mathsf{T}'\rangle.T, c' : \mathsf{T}'} \ (\textsc{DelegT}) \qquad \frac{\Gamma \vdash \mathsf{P} \triangleright \Delta, c : T, y : \mathsf{T}}{\Gamma \vdash c?((\mathsf{q}, y)).\mathsf{P} \triangleright \Delta, c : ?(\mathsf{q}, \mathsf{T}).T} \ (\textsc{ChRcvT})$$

$$\frac{\Gamma \vdash \mathsf{P} \triangleright \Delta, c : T_j \qquad j \in I}{\Gamma \vdash c \oplus \langle \mathsf{p}, \mathsf{P}\rangle \triangleright \Delta, c : \oplus\langle \mathsf{p}, \{l_i : T\}_{i \in I}\rangle} \ (\textsc{LSndT}) \qquad \frac{\Gamma \vdash \mathsf{P}_i \triangleright \Delta, c : T_i \qquad \forall i \in I}{\Gamma \vdash c\&(\mathsf{p}, \{l_i : \mathsf{P}\}_{i \in I}) \triangleright \Delta, c : \&(\mathsf{p}, \{l_i : T\}_{i \in I})} \ (\textsc{BranchT})$$

$$\frac{\Gamma \vdash \mathsf{P} \triangleright \Delta \qquad \Gamma \vdash \mathsf{Q} \triangleright \Delta'}{\Gamma \vdash \mathsf{P} \parallel \mathsf{Q} \triangleright \Delta, \Delta'} \ (\textsc{ParT}) \qquad \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash \mathsf{P} \triangleright \Delta \qquad \Gamma \vdash \mathsf{Q} \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q} \triangleright \Delta} \ (\textsc{IfT})$$

$$\frac{\Delta\ \mathsf{End}\ \text{only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \ (\textsc{ZeroT}) \qquad \frac{\Gamma, X : \Delta \vdash \mathsf{P} \triangleright \Delta}{\Gamma \vdash \mu \mathsf{t}.\mathsf{P} \triangleright \Delta} \ (\textsc{RecT}) \qquad \Gamma, \mathsf{X} : \Delta \vdash \mathsf{X} \triangleright \Delta \ (\textsc{ProcVarT})$$

Figure 1.3: Typing rules

and the session types obtained as projections of G:

$$\begin{aligned}
\mathsf{T}_\mathsf{s} &= ?(1, \mathtt{str}).!\langle 2, \mathtt{int}\rangle.!\langle 1, \mathtt{int}\rangle.\mathsf{T}'_\mathsf{s} \\
\mathsf{T}_1 &= !\langle 3, \mathtt{str}\rangle.?(3, \mathtt{int}).!\langle 2, \mathtt{int}\rangle.\mathsf{End} \\
\mathsf{T}_2 &= ?(3, \mathtt{int}).?(1, \mathtt{int}).\mathsf{T}'_2
\end{aligned}$$

Let $\Gamma_\mathsf{s} = a : \mathsf{G}, x : \mathtt{str}$ and $\Gamma_1 = a : \mathsf{G}, x : \mathtt{int}$. Type derivations for the processes modeling the "Two Buyer" example are given in Fig. 1.4.

$$\dfrac{a : \mathsf{G} \vdash a : \mathsf{G} \quad \dfrac{a : \mathsf{G} \vdash y?(1,x).y!\langle 2,10\rangle.y!\langle 1,10\rangle.P_{\mathsf{s}} \triangleright y :?(1,\mathtt{str}).!\langle 2,\mathtt{int}\rangle.!\langle 1,\mathtt{int}\rangle.\mathsf{T}'_{\mathsf{s}} \quad \dfrac{\Gamma_{\mathsf{s}} \vdash 10 : \mathtt{int} \quad \dfrac{\Gamma_{\mathsf{s}} \vdash 10 : \mathtt{int} \quad \dfrac{\Gamma_{\mathsf{s}} \vdash P_{\mathsf{s}} \triangleright T_{\mathsf{s}}}{\Gamma_{\mathsf{s}} \vdash y!\langle 1,10\rangle.P_{\mathsf{s}} \triangleright y : !\langle 1,\mathtt{int}\rangle.\mathsf{T}'_{\mathsf{s}}}}{\Gamma_{\mathsf{s}} \vdash y!\langle 2,10\rangle.y!\langle 1,10\rangle.P_{\mathsf{s}} \triangleright y : !\langle 2,\mathtt{int}\rangle.!\langle 1,\mathtt{int}\rangle.\mathsf{T}'_{\mathsf{s}}}}{a : \mathsf{G} \vdash \overline{a}[3](y).y?(1,x).y!\langle 2,10\rangle.y!\langle 1,10\rangle.P_{\mathsf{s}} \triangleright \emptyset}$$

$$\dfrac{a : \mathsf{G} \vdash a : \mathsf{G} \quad \dfrac{a : \mathsf{G} \vdash \mathit{title} : \mathtt{str} \quad \dfrac{a : \mathsf{G} \vdash y?(3,x).y!\langle 2,x/2\rangle.\mathbf{0} \triangleright y :?(3,\mathtt{int}).!\langle 2,\mathtt{int}\rangle.\mathsf{End} \quad \dfrac{\Gamma_1 \vdash x/2 : \mathtt{int} \;\; \Gamma_1 \vdash \mathbf{0} \triangleright y : \mathsf{End}}{\Gamma_1 \vdash y!\langle 2,x/2\rangle.\mathbf{0} \triangleright y :!\langle 2,\mathtt{int}\rangle.\mathsf{End}}}{a : \mathsf{G} \vdash y!\langle 3,\mathit{title}\rangle.y?(3,x).y!\langle 2,x/2\rangle.\mathbf{0} \triangleright y :!\langle 3,\mathtt{str}\rangle.?(3,\mathtt{int}).!\langle 2,\mathtt{int}\rangle.\mathsf{End}}}{a : \mathsf{G} \vdash a[1](y).y!\langle 3,\mathit{title}\rangle.y?(3,x).y!\langle 2,x/2\rangle.\mathbf{0} \triangleright \emptyset}$$

$$\dfrac{a : \mathsf{G} \vdash a : \mathsf{G} \quad \dfrac{a : \mathsf{G} \vdash y?(3,x).y?(1,x_1).P_2 \triangleright y :?(3,\mathtt{int}).?(1,\mathtt{int}).\mathsf{T}'_2 \quad \dfrac{a : \mathsf{G}, x : \mathtt{int} \vdash y?(1,x_1).P_2 \triangleright y : ?(1,\mathtt{int}).\mathsf{T}'_2 \quad \dfrac{a : \mathsf{G}, x : \mathtt{int}, x_1 : \mathtt{int} \vdash P_2 \triangleright y : \mathsf{T}'_2}{\vdots}}{a : \mathsf{G}, x : \mathtt{int} \vdash y?(1,x_1).P_2 \triangleright y : ?(1,\mathtt{int}).\mathsf{T}'_2}}{a : \mathsf{G} \vdash y?(3,x).y?(1,x_1).P_2 \triangleright y :?(3,\mathtt{int}).?(1,\mathtt{int}).\mathsf{T}'_2}}{a : \mathsf{G} \vdash a[2](y).y?(3,x).y?(1,x_1).P_2 \triangleright \emptyset}$$

Figure 1.4: (Part of) type derivations for the processes of the example.

Chapter 2

Coinduction and co-logic programming

In this chapter we introduce the fundamental ingredients of the coinductive approach used in the thesis work. First, in Sect. 2.1 we recall the notion of coinductive definition, as opposed to inductive definition, in the framework of inference systems. Coinductive definitions will be extensively used in Chapter 3. Then, in Sect. 2.2, we recall basic notions of logic programming, and how the classic paradigm can be extended to support coinductive predicates. The implementation we have developed, described in Chapter 4, uses SWI-Prolog, a Prolog environment where predicates can be marked as coinductive.

2.1 Inference systems

A very used and comprehensible way to express inductive and coinductive definitions is by *inference systems*. An inference system is a set of rules, describing how we can derive a new object/judgment (consequence) starting from some premises.

Formally, let $\mathcal{U}$ be a set, called *universe*, whose elements are called *judgments*. An *inference rule* is a pair $\dfrac{Pr}{c}$, where $Pr \subseteq \mathcal{U}$ is a set whose elements are called *premises*, $c \in \mathcal{U}$ is called *consequence*. We call *inference system* a set $\mathcal{I}$ of inference rules. A rule whose set of premises is empty is called *axiom*. It is customary to define an infinite set of rules in a finitary way using *meta-rules*. For example, taken as universe the set of natural numbers, the rules that define the set of even numbers are infinite, but can described using the following two meta-rules, where the meta-variable n ranges over the universe.

$$\frac{}{0} \qquad\qquad \frac{n}{n+2}$$

An inference system defines a set of judgments; in particular, two possible interpretations are possible: the *inductive* and the *coinductive interpretation*. These definitions are based on the notions of *closed* and *consistent* set. Given $\mathcal{S} \subseteq \mathcal{U}$, $\mathcal{S}$ is *closed* if, for all $\frac{Pr}{c} \in \mathcal{I}$, $Pr \subseteq \mathcal{S}$ implies $c \in \mathcal{S}$; $\mathcal{S}$ is *consistent* if, for all $x \in \mathcal{S}$, there exists $\frac{Pr}{x} \in \mathcal{I}$ such that $Pr \subseteq \mathcal{S}$. Then:

- The inductive interpretation of $\mathcal{I}$, denoted $Ind(\mathcal{I})$, is the smallest closed set, that is, the intersection of all the closed sets.
- The coinductive interpretation of $\mathcal{I}$, denoted $CoInd(\mathcal{I})$, is the largest consistent set, that is, the union of all the consistent sets[1].

The inductive and coinductive interpretations can also be characterized in terms of proof trees. That is, defining a proof tree in $\mathcal{I}$ as a tree whose nodes are (labeled with) judgments in $\mathcal{U}$, and there is a node c with set of children Pr only if $\frac{Pr}{c} \in \mathcal{I}$, it can be shown [LG09] that $Ind(\mathcal{I})$ and $CoInd(\mathcal{I})$ are the sets of judgments which are the root of a finite[2]and an arbitrary (finite or infinite) proof tree, respectively. From this definition we derive that $Ind(\mathcal{I}) \subseteq CoInd(\mathcal{I})$.

Typically, we want to prove that a given set $\mathcal{S}$ (the expected semantics) can be defined either inductively, that is, as $Ind(\mathcal{I})$, or coinductively, that is, as $CoInd(\mathcal{I})$, by some inference system $\mathcal{I}$.

For a set defined inductively, the *induction principle* provides a technique to prove that $Ind(\mathcal{I}) \subseteq \mathcal{S}$, that is, that the inductive definition is *sound* with respect to $\mathcal{S}$.

For a set defined coinductively, the *coinduction principle* provides a technique to prove that $\mathcal{S} \subseteq CoInd(\mathcal{I})$, that is, that the coinductive definition is *complete* with respect to the $\mathcal{S}$.

The two proof principles are stated below.

- Given a closed set $\mathcal{S} \subseteq \mathcal{U}$, $Ind(\mathcal{I}) \subseteq \mathcal{S}$ (Induction principle).

[1]It can be proved that an intersection of closed sets is closed, and an union of consistent sets is consistent

[2]Under the common assumption that the set of premises of all the rules are finite, otherwise we should say a finite depth tree.

- Given a consistent set $\mathcal{S} \subseteq \mathcal{U}$, $\mathcal{S} \subseteq \mathit{CoInd}(\mathcal{I})$ (Coinduction principle).

Consider, for instance, the predicate allPos on (possibly infinite) lists of integers, such that $\mathsf{allPos}(l)$ holds iff all the elements of the list l are positive, and the following inference system:

$$\frac{}{\mathsf{allPos}(\Lambda)} \qquad \frac{\mathsf{allPos}(l)}{\mathsf{allPos}(x\colon l)}\ x > 0$$

where Λ represents the empty list and $\_ : \_$ represents the concatenation of an element to a list. In this example, the inductive interpretation contains all the judgments $\mathsf{allPos}(l)$ such that l is finite and all its elements are positive. On the other side, there is no proof tree if l is infinite. If we consider the coinductive interpretation, instead, then the judgment is derivable, with an infinite proof tree, for infinite lists with all positive elements. For instance, the following infinite proof trees prove that the list of all odd natural numbers, and the list `[1,2,1,2,...]`, respectively, contain only positive elements:

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{allPos}(5:7:9:\ldots)}}{\mathsf{allPos}(3:5:7:\ldots)}}{\mathsf{allPos}(1:3:5\ldots)} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\mathsf{allPos}(1:2:1:\ldots)}}{\mathsf{allPos}(2:1:2:\ldots)}}{\mathsf{allPos}(1:2:1\ldots)}$$

Hence, in this case, the expected semantics on possibly infinite lists is provided by the coinductive interpretation. However, this is not always the case: it is easy to see that for the predicate member such that $\mathsf{member}(x, l)$ holds iff x is an element of l, the following inference system

$$\frac{}{\mathsf{member}(x, x\colon \Lambda)} \qquad \frac{\mathsf{member}(x, l)}{\mathsf{member}(x, y\colon l)}\ x \neq y$$

should be interpreted inductively, since otherwise the judgment $\mathsf{member}(x, l)$ could be always derived for l infinite list. These examples show that, when working with possibly infinite structures, both inductive and coinductive definitions are needed, as actually supported by co-logic programming.

We conclude this section by a very important remark, concerning the difference between the two proof trees shown above. Indeed, whereas both are infinite, the latter is *regular*, that is, has a finite number of subtrees. In other words, it requires only the proof of *finitely many* different judgements. In this case, it is possible to design an algorithm, checking whether a judgement can be derived, which successfully terminates on the derivable judgements. The algorithm keeps track of already encountered judgements and, if the same judgement is found again, we can use it as an axiom. The same idea is used in the operational semantics of coinductive logic programming, see next section.

2.2 Co-logic programming

In this section we recall basic concepts about logic programming, and how the classic paradigm can be extended to support coinductive predicates.

A signature consists in a set of *predicate symbols* p, *function symbols* f and *variables* X, each one associated with an *arity* ≥ 0. Variables have arity 0. A function with arity 0 is called *constant*. A *term* is a tree whose nodes are labeled with functions and variable symbols, so that the number of children is the arity. An *atom* is a tree whose nodes are labeled with a predicate symbol and other nodes are labeled with function and variable symbols. Terms and atoms are *ground* if they do not contain variables, and *finite* (or *syntactic*) if they are finite trees. A *logic program* is a set of *clauses*, of shape A :- $B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are finite atoms. A clause where $n = 0$ is called a *fact*.

A *substitution* θ is a mapping from a finite subset of variables into terms. We write $t\theta$ for the application of a substitution θ to a term t, and call $t\theta$ an *instance* of t. These notions can be analogously defined on atoms and clauses. A substitution is *ground* if it maps variables into ground terms, *syntactic* if it maps variables into finite (syntactic) terms.

The *declarative semantics* of a logic program describes its meaning in an abstract way, as the set of ground atoms which are defined to be true by the program, in a sense to be made precise depending on the kind of declarative semantics we choose, as detailed below.

The *Herbrand universe* HU is defined as the set of finite ground terms, and the *Herbrand base* HB as the set of finite ground atoms. Sets $I \subseteq \mathsf{HB}$ are called *interpretations*. Given a logic program P, we can define the *inference operator* $T_P : \wp(\mathsf{HB}) \to \wp(\mathsf{HB})$ as follows:

$$T_P(I) = \{A \mid (A \text{ :- } B_1, \ldots, B_n) \in \mathsf{ground}(P), \{B_1, \ldots, B_n\} \subseteq I\}$$

where $\mathsf{ground}(P)$ is the set of instances of clauses in P obtained by a ground syntactic substitution.

An interpretation is a *model* of a program P (is *closed* with respect to P) if $T_P(I) \subseteq I$. The standard declarative semantics *Ind*(P) of P is the least interpretation which is a model taking as order set inclusion, that is, the intersection of all closed interpretations. Defining a *proof tree* for a ground atom A as a tree where the root is A, nodes are ground instances of rules, and leaves are ground instances of facts, the standard declarative semantics can be equivalently characterized as the set of finite ground atoms which have a finite proof tree.

The *operational semantics* of a logic program is an effective procedure which, given a *goal* of shape $? - A_1, \ldots, A_n$, finds its solutions, represented by substitutions.

The standard operational semantics of a logic program is *SLD-resolution*, which, at each step, selects an atom A from the current goal, looks for a clause A' `:-` $B_1, \ldots, B_n$ in the program whose head unifies with the selected atom, replaces A by atoms $B_1, \ldots, B_n$ in the current goal and applies the substitution deriving from the unification. These steps are iterated until getting an empty goal.

A limit of the standard declarative semantics described above is that we cannot define predicates on non-well-founded structures, such as infinite lists. To overcome this, logic programming can be extended to support coinduction by *coinductive logic programming* [SMBG06b, SMBG06a, AD15], where terms are coinductively defined, that is, can be infinite, and predicates are coinductively defined as well. Possibly infinite terms are represented by finite sets of equations between finite terms. For instance, the equation `L` $=$ `[1,2|L]` represents the infinite list `[1,2,1,2,...]`. On the other hand, the infinite list of odd numbers *cannot* be represented by a finite set of equations. The following logic program defines the predicate allPos described in Sect. 2.1.

```
allPos([]).
allPos([N|L]) :- N>0, allPos(L).
```

To define the declarative semantics, first of all infinite terms and atoms should be included. The *complete Herbrand universe* co-HU is the set of (finite and infinite) ground terms. The *complete Herbrand base* co-HB is the set of (finite and infinite) ground atoms. Sets $I \subseteq$ co-HB are called *co-interpretations*.

We can define the inference operator $T_P : \wp(\text{co-HB}) \to \wp(\text{co-HB})$ analogously to that above:

$$T_P(I) = \{A \mid (A \texttt{ :- } B_1, \ldots, B_n) \in \text{co-ground}(P), \{B_1, \ldots, B_n\} \subseteq I\}$$

where co-ground(P) is the set of instances of clauses in P obtained by a ground substitution.

A co-interpretation I is a *co-model* of P (is *consistent* with respect to P) if and only if $I \subseteq T_P(I)$. The *coinductive (declarative) semantics* of P, denoted $CoInd(P)$, is the greatest co-interpretation which is a co-model taking as order set inclusion, that is, the union of all consistent co-interpretations. Equivalently, $CoInd(P)$ can be characterized as the set of ground atoms which have a (finite or infinite) proof tree.

As the reader may have note, the above definitions can be seen as a particular case of those given in Sect. 2.1, since the clauses of a logic program can be seen as meta-rules of an inference system where judgments are ground atoms.

In coinductive logic programming, standard SLD resolution is replaced by co-SLD resolution [SMBG06a, AD15], which, roughly speaking, keeps trace of the already encountered goals, called *(dynamic) coinductive hypotheses*, so that, when a goal is found the second time, it is considered successful. In this way, for instance, resolution can give a positive answer to the goal $\langle$`allPos(L)`$; \{$`L` $=$ `[1,2|L]`$\}\rangle$.

A drawback of coinductive logic programming is that *all* predicates are interpreted coinductively, whereas in applications it is often the case that predicates to be interpreted either inductively and coinductively should coexist. To overcome this issue, *co-logic programming* [SBMG07] marks predicates as either inductive or coinductive; however, no mutual recursion is allowed between an inductive and a coinductive predicate, that is, stratification is needed. This approach of marking predicates is supported by SWI-Prolog, the Prolog environment we have used for implementation, as described in Chapter 4. Thanks to the constraint that no mutual recursion is allowed, at each layer we only have either only inductive or only coinductive predicates. Hence each layer can be interpreted as the least or greatest fixed point, respectively, of an inference system where the lower levels are assumed as axioms.

Chapter 3

Global types for asynchronous networks

In this chapter, we describe a proposal, currently under development, for a new formalism of global types. The main difference with respect to the classical approach described in Chapter 1 is that output and input operations are specified separately. This feature makes the global types more "low-level", hence able to directly handle asynchrony at the level of the type system, because the output and the corresponding input operations are not forced to happen at the same time. Instead, usual global types only specify communications, hence, when used to describe asynchronous networks, they have to be combined with a subtyping relation on session types, firstly proposed in [MYH09a]. Unfortunately, this subtyping has been shown to be undecidable [BCZ17a, LY17].

The other novelty with respect to the classical presentation is that a *coinductive approach* is adopted. Namely, processes and types with an infinite behaviour are expressed as infinite regular terms, rather than by an explicit fixed point operator, and, correspondingly, functions handling them, e.g., the projection, are also defined coinductively. Such coinductive approach will be exploited in Chapter 4 to provide an implementation in co-logic programming.

The proposed formalism, to keep the focus on the key new feature, which is explicitly asynchronous communication, considers a greatly simplified calculus with respect to that presented in Chapter 1. Notably, a global type describes a single session, so in the calculus there is no notion of session, hence no hiding, and channels are implicit, one for each pair of participants. Moreover, there is no information exchange among participants; they can only send labels, used to have conditional branches in the behaviour. In this way, it is possible to directly obtain processes as projections of global types, without having to explicitly introduce session types, see Sect. 1.3. Finally, delegation, parallel operator,

and conditional are omitted, whereas recursion is not needed, since recursive processes are obtained as regular coinductive terms, see below.

In Sect. 3.1 we present the process calculus, in Sect. 3.2 the global types, and in Sect. 3.3 projection and type checking.

3.1 Process calculus

We assume base sets of *participants* $\mathsf{p}, \mathsf{q}, \mathsf{r} \in \mathsf{Part}$, and *labels* $\lambda \in \mathsf{Lab}$. The syntax of processes is as follows:

$$\mathsf{P} \quad ::=_\rho \quad \mathsf{p}!\{\lambda_i.\mathsf{P}_i\}_{i\in I} \mid \mathsf{p}?\{\lambda_i.\mathsf{P}_i\}_{i\in I} \mid \mathbf{0}$$

where $I \neq \emptyset$, and $\lambda_j \neq \lambda_h$ for $j \neq h$. The symbol $::=_\rho$, in the definition above and others in the following, indicates that the productions should be interpreted *coinductively*, rather than inductively as in the standard case. That is, they define possibly infinite terms (processes in the case above). However, we assume such terms to be *regular*, that is, with finitely many distinct sub-terms. In this way, we only obtain terms (processes in the case above) which are solutions of a finite set of equations, see [Cou83].

A process of shape $\mathsf{p}!\{\lambda_i.\mathsf{P}_i\}_{i\in I}$ (*internal choice*) chooses a label in $\{\lambda_i \mid i \in I\}$ to be sent to p, and then behaves differently depending on the sent label. A process of shape $\mathsf{p}?\{\lambda_i.\mathsf{P}_i\}_{i\in I}$ (*external choice*) waits for receiving one of the labels $\{\lambda_i \mid i \in I\}$ from p, and then behaves differently depending on the received label. Note that the set of indexes in choices is assumed to be non-empty, and the corresponding labels to be all different.

An internal choice which is a singleton is simply written $\mathsf{p}!\lambda.\mathsf{P}$, and $\mathsf{p}!\lambda.\mathbf{0}$ is abbreviated $\mathsf{p}!\lambda$; analogously for an external choice.

As in the calculus of Chapter 1, *messages* are triples $\langle \mathsf{p}, \lambda, \mathsf{q}\rangle$, denoting that participant p has sent label λ to participant q. Accordingly with the simplification that there are no sessions, there is a single queue, $\mathcal{M}$, defined as before:

$$\mathcal{M} ::= \emptyset \mid \langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \mathcal{M}$$

Since the only order that matters is that between messages with the same sender and receiver, the same structural equivalence of Chapter 1 holds:

$$\mathcal{M} \cdot \langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \langle \mathsf{r}, \lambda', \mathsf{s}\rangle \cdot \mathcal{M}' \equiv \mathcal{M} \cdot \langle \mathsf{r}, \lambda', \mathsf{s}\rangle \cdot \langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \mathcal{M}' \quad \text{if} \quad \mathsf{p} \neq \mathsf{r} \quad \text{or} \quad \mathsf{q} \neq \mathsf{s}$$

Note that $\langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \langle \mathsf{q}, \lambda', \mathsf{p}\rangle \equiv \langle \mathsf{q}, \lambda', \mathsf{p}\rangle \cdot \langle \mathsf{p}, \lambda, \mathsf{q}\rangle$. This is the situation in which both participants p and q have sent a message to the other one, and neither of them has read the message, as it could happen in a network with asynchronous communication. As noted

in Sect. 1.2, an equivalent view of the queue is as a map from pairs of participants to lists of labels.

Networks are composed of at least two components of shape $\mathsf{p}[\![\,\mathsf{P}\,]\!]$ in parallel, each with a different participant p, and a message queue. That is, a network has shape:

$$\mathbb{N} \parallel \mathcal{M}$$

where

$$\mathbb{N} ::= \mathsf{p}_1[\![\,\mathsf{P}_1\,]\!] \parallel \cdots \parallel \mathsf{p}_n[\![\,\mathsf{P}_n\,]\!] \qquad n \geq 2,\ \mathsf{p}_i \neq \mathsf{p}_j \ \text{ for } \ i \neq j$$

We can define the sets of *participants* of networks and queues:

$$\begin{aligned}
&\mathsf{part}(\mathsf{p}_1[\![\,\mathsf{P}_1\,]\!] \parallel \cdots \parallel \mathsf{p}_n[\![\,\mathsf{P}_n\,]\!]) = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\} \\
&\mathsf{part}(\emptyset) = \emptyset \\
&\mathsf{part}(\langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \mathcal{M}) = \{\mathsf{p}, \mathsf{q}\} \cup \mathsf{part}(\mathcal{M})
\end{aligned}$$

3.2 Global types

The new definition of *global types* is as follows:

$$\mathsf{G} \ ::=_{\rho} \ \mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I} \mid \mathsf{pq}?\lambda.\mathsf{G} \mid \mathsf{End}$$

where $I \neq \emptyset$, $\mathsf{p} \neq \mathsf{q}$, and $\lambda_j \neq \lambda_h$ for $j \neq h$. The basic difference, as said above, is that output and input operations are specified separately. Moreover, accordingly with the fact that value passing and delegation are omitted in the calculus, there are no exchange types. Finally, as for processes, these global types are defined coinductively, so that infinite global types are allowed, but only of regular shape.

When I is a singleton, we simply write $\mathsf{pq}!\lambda.\mathsf{G}$. Moreover, $\mathsf{pq}!\lambda.\mathsf{End}$ and $\mathsf{pq}?\lambda.\mathsf{End}$ are abbreviated $\mathsf{pq}!\lambda$ and $\mathsf{pq}!\lambda$, respectively.

A pair $\mathsf{G} \parallel \mathcal{M}$ where G is a global type and $\mathcal{M}$ is a message queue is called *configuration type*. The function part associates to global types their sets of *participants*, which are the smallest sets such that:

$$\begin{aligned}
\mathsf{part}(\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}) &= \{\mathsf{p}, \mathsf{q}\} \cup \textstyle\bigcup_{i\in I} \mathsf{part}(\mathsf{G}_i) \\
\mathsf{part}(\mathsf{pq}?\lambda.\mathsf{G}) &= \{\mathsf{p}, \mathsf{q}\} \cup \mathsf{part}(\mathsf{G}) \\
\mathsf{part}(\mathsf{End}) &= \emptyset
\end{aligned}$$

The function player associates to global types their sets of *players*, that is, participants

that are active, which are the smallest sets such that:

$$\begin{array}{lcl} \mathsf{play}(\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}) & = & \{\mathsf{p}\} \cup \bigcup_{i\in I} \mathsf{play}(\mathsf{G}_i) \\ \mathsf{play}(\mathsf{pq}?\lambda.\mathsf{G}) & = & \{\mathsf{q}\} \cup \mathsf{play}(\mathsf{G}) \\ \mathsf{play}(\mathsf{End}) & = & \emptyset \end{array}$$

It is easy to see that, due to the regularity assumption on global types, the sets of participants and players are finite.

A *path* ξ in a global type G is a possibly infinite sequence of *communications*, of shape either $\mathsf{pq}!\lambda$ or $\mathsf{pq}?\lambda$. Set $|\xi| \in \mathbf{N} \cup \{\omega\}$ the length of ξ. We denote by ξ_n the n-th communication in the path ξ, with $0 \leq n < |\xi|$, by ϵ the empty sequence, and by $\cdot$ the concatenation of a finite sequence with a possibly infinite sequence. Set $\mathsf{play}(\mathsf{pq}!\lambda) = \mathsf{p}$ and $\mathsf{play}(\mathsf{pq}?\lambda) = \mathsf{q}$. Then, $\mathsf{play}(\xi) = \bigcup_{0\leq n<|\xi|} \mathsf{play}(\xi_n)$.

The function Paths associates to global types their sets of *paths*, which are the greatest sets such that:

$$\begin{array}{lcl} \mathsf{Paths}(\mathsf{End}) & = & \{\epsilon\} \\ \mathsf{Paths}(\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}) & = & \bigcup_{i\in I}\{\mathsf{pq}!\lambda_i \cdot \xi \mid \xi \in \mathsf{Paths}(\mathsf{G}_i)\} \\ \mathsf{Paths}(\mathsf{pq}?\lambda.\mathsf{G}) & = & \{\mathsf{pq}?\lambda \cdot \xi \mid \xi \in \mathsf{Paths}(\mathsf{G})\} \end{array}$$

It is easy to check that $\mathsf{play}(\mathsf{G}) = \bigcup_{\xi\in\mathsf{Paths}(\mathsf{G})} \mathsf{play}(\xi)$.

To enforce by typing the property that a participant cannot wait forever, we have to restrict ourselves to a subset of "well-behaved" global types, by imposing an additional syntactic condition: the first occurrences of participants as players in a global type are at a bounded depth in all paths starting from the root. A global type with this property is called *fair*. This property is formalised by the following definition. For $\xi \in \mathsf{Paths}(\mathsf{G})$, set $\mathsf{depth}(\xi, \mathsf{p}) = \inf\{n \mid \mathsf{play}(\xi_n) = \mathsf{p}\}$, and define $\mathsf{depth}(\mathsf{G}, \mathsf{p})$, the *depth* of p in G, as follows:

$$\mathsf{depth}(\mathsf{G}, \mathsf{p}) = \begin{cases} 1 + \sup\{\mathsf{depth}(\xi, \mathsf{p}) \mid \xi \in \mathsf{Paths}(\mathsf{G})\} & \text{if } \mathsf{p} \in \mathsf{play}(\mathsf{G}) \\ 0 & \text{otherwise} \end{cases}$$

Note that, if $\mathsf{p} \notin \mathsf{play}(\xi)$ for some path ξ, then $\mathsf{depth}(\xi, \mathsf{p}) = \inf \emptyset = \infty$. Hence, if p is a player of a global type G, but it does not occur as a player in some path of G, then $\mathsf{depth}(\mathsf{G}, \mathsf{p}) = \infty$, modelling the fact that p may wait forever. For example, in the global type $\mathsf{G} = \mathsf{pq}!\{\lambda_1.\mathsf{pq}?\lambda_1; \mathsf{rq}!\lambda_3, \lambda_2.\mathsf{pq}?\lambda_2.\mathsf{G}\}$, the player r does not occur in the infinite path $\xi = \mathsf{pq}!\lambda_2 \cdot \mathsf{pq}?\lambda_2 \cdot \mathsf{pq}!\lambda_2 \cdot \mathsf{pq}?\lambda_2 \cdot \ldots$.

We can now formally define the fairness property introduced above: a global type G is *fair* if, for all participants $\mathsf{p} \in \mathsf{play}(\mathsf{G})$ and subterm G' of G, $\mathsf{depth}(\mathsf{G}', \mathsf{p})$ is finite.

3.3 Projection and type checking

Projection Configuration types are projected as in Sect. 1.4. However, due to the simplification that there is no level of session types, projecting a configuration type we directly get a process.

The fact that projecting $\mathsf{G} \parallel \mathcal{M}$ on a participant p we get a process P is modeled by the judgment $\langle (\mathsf{G} \parallel \mathcal{M}) \upharpoonright \mathsf{p}, \mathsf{P} \rangle$, defined by the rules in Fig. 3.1. The thick line indicates that such rules should be interpreted coinductively, rather than inductively as in the standard case, so we allow possibly infinite proof trees. However, we assume such proof trees to be *regular*, that is, with finitely many distinct sub-trees.

The projection definition uses process contexs with an arbitrary number of holes indexed with natural numbers. We assume that each hole has a different index. Given a context $\mathcal{C}$ with holes indexed in J, we denote by $\mathcal{C}[\mathsf{P}_j]_{j \in J}$ the process obtained by filling the hole indexed by j by P_j, for all $j \in J$. The contexts have the following syntax:

$$\mathcal{C} ::= [\,]_n \quad | \quad \mathsf{P} \mid \quad \mathsf{p}!\{\lambda_i.\mathcal{C}_i\}_{i \in I} \quad | \quad \mathsf{p}?\{\lambda_i.\mathcal{C}_i\}_{i \in I} \qquad \text{where } I \neq \emptyset, \text{ and } \lambda_j \neq \lambda_h \text{ for } j \neq h$$

Rule (Ext) states that projecting on a participant which is not a player gives the inactive process.

The following two rules describe the effect of projecting a configuration type where the global type starts with the reception by participant q of the label λ sent by participant p, and continues as G. In rule (In-Rcv), projecting on the player q (the receiver) is only possible if λ is actually the label of the first message from p to q in the queue, and gives the process waiting for the input λ from p, and then continuing as the projection of G and the queue where the message has been read. In rule (In-Ext), projecting on another participant which is a player in G simply gives the projection of G and the queue where the message has been read. Note that the case where the participant is not a player in G is covered by rule (Ext).

The following three rules describe the effect of projecting a configuration type where the global type starts with sending a label, chosen in a set, from participant p to participant q, and continues, depending on the chosen label λ_i, as G_i.

Rule (Out-Snd) is symmetrical to rule (In-Rcv). That is, projecting on the player p (the sender) gives the process (an external choice) sending a label in the given set to q, and then continuing, for each chosen label λ_i, as the projection of G_i and the queue where the corresponding message has been added. Moreover, in order to ensure that there are no *orphan-messages*, that is, messages that are not eventually read, the added message should be actually read by each G_i, as required by the side-condition, where:

- ζ is a (finite) sequence of labels

$$\text{(Ext)}\ \frac{}{\langle (\mathsf{G} \parallel \mathcal{M}) \upharpoonright \mathsf{p}\,, \mathbf{0}\rangle}\ \mathsf{p} \notin \mathsf{play}(\mathsf{G})$$

$$\text{(In-Rcv)}\ \frac{\langle (\mathsf{G} \parallel \mathcal{M}) \upharpoonright \mathsf{q}\,, \mathsf{P}\rangle}{\langle (\mathsf{pq}?\lambda.\mathsf{G} \parallel \langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \mathcal{M}) \upharpoonright \mathsf{q}\,, \mathsf{p}?\lambda.\mathsf{P}\rangle}$$

$$\text{(In-Ext)}\ \frac{\langle (\mathsf{G} \parallel \mathcal{M}) \upharpoonright \mathsf{s}\,, \mathsf{P}\rangle}{\langle (\mathsf{pq}?\lambda.\mathsf{G} \parallel \langle \mathsf{p}, \lambda, \mathsf{q}\rangle \cdot \mathcal{M}) \upharpoonright \mathsf{s}\,, \mathsf{P}\rangle}\ \begin{array}{l}\mathsf{s} \neq \mathsf{q}\\ \mathsf{s} \in \mathsf{play}(\mathsf{G})\end{array}$$

$$\text{(Out-Snd)}\ \frac{\langle (\mathsf{G}_i \parallel \mathcal{M} \cdot \langle \mathsf{p}, \lambda_i, \mathsf{q}\rangle) \upharpoonright \mathsf{p}\,, \mathsf{P}_i\rangle \quad \forall i \in I}{\langle (\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I} \parallel \mathcal{M}) \upharpoonright \mathsf{p}\,, \mathsf{q}!\{\lambda_i.\mathsf{P}_i\}_{i\in I}\rangle}\ \vdash_{\mathsf{read}} \langle \mathsf{G}_i, \mathsf{p}, \mathsf{q}, \mathcal{M}\!\upharpoonright\!(\mathsf{p},\mathsf{q}) \cdot \lambda_i\rangle\ \forall i \in I$$

$$\text{(Out-Rcv)}\ \frac{\langle (\mathsf{G}_i \parallel \mathcal{M} \cdot \langle \mathsf{p}, \lambda_i, \mathsf{q}\rangle) \upharpoonright \mathsf{q}\,, \mathcal{C}[\mathsf{p}?\lambda_i.\mathsf{P}_{i,j}]_{j\in J}\rangle \quad \forall i \in I}{\langle (\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I} \parallel \mathcal{M}) \upharpoonright \mathsf{q}\,, \mathcal{C}[\mathsf{p}?\{\lambda_i.\mathsf{P}_{i,j}\}_{i\in I}]_{j\in J}\rangle}$$

$$\text{(Out-Ext)}\ \frac{\langle (\mathsf{G}_i \parallel \mathcal{M} \cdot \langle \mathsf{p}, \lambda_i, \mathsf{q}\rangle) \upharpoonright \mathsf{s}\,, \mathcal{C}[\mathsf{r}?\lambda'_i.R_{i,j}]_{j\in J}\rangle \quad \forall i \in I}{\langle (\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I} \parallel \mathcal{M}) \upharpoonright \mathsf{s}\,, \mathcal{C}[\mathsf{r}?\{\lambda'_i.R_{i,j}\}_{i\in I}]_{j\in J}\rangle}\ \begin{array}{l}\mathsf{s} \notin \{\mathsf{p},\mathsf{q}\}\\ \mathsf{s} \in \mathsf{play}(\mathsf{G}_i)\ i \in I\end{array}$$

$$\frac{}{\vdash_{\mathsf{read}} \langle \mathsf{G}, \mathsf{p}, \mathsf{q}, \epsilon\rangle} \qquad \frac{\vdash_{\mathsf{read}} \langle \mathsf{G}_i, \mathsf{p}, \mathsf{q}, \zeta\rangle \quad i \in I}{\vdash_{\mathsf{read}} \langle \mathsf{rs}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}, \mathsf{p}, \mathsf{q}, \zeta\rangle}$$

$$\frac{\vdash_{\mathsf{read}} \langle \mathsf{G}, \mathsf{p}, \mathsf{q}, \zeta\rangle}{\vdash_{\mathsf{read}} \langle \mathsf{pq}?\lambda.\mathsf{G}, \mathsf{p}, \mathsf{q}, \lambda \cdot \zeta\rangle} \qquad \frac{\vdash_{\mathsf{read}} \langle \mathsf{G}, \mathsf{p}, \mathsf{q}, \zeta\rangle}{\vdash_{\mathsf{read}} \langle \mathsf{rs}?\lambda.\mathsf{G}, \mathsf{p}, \mathsf{q}, \zeta\rangle}\,\mathsf{r} \neq \mathsf{p} \text{ or } \mathsf{s} \neq \mathsf{q}$$

Figure 3.1: Projection

$$\text{(T-Net)}\ \frac{\langle (\mathsf{G} \parallel \mathcal{M}) \upharpoonright \mathsf{p}_i , \mathsf{P}'_i \rangle \quad i \in 1..n \qquad \mathsf{P}_i \leq \mathsf{P}'_i \quad i \in 1..n \qquad \mathsf{part}(\mathsf{G}) \subseteq \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}}{\vdash \mathsf{p}_1[\![\mathsf{P}_1]\!] \parallel \cdots \parallel \mathsf{p}_n[\![\mathsf{P}_n]\!] \parallel \mathcal{M} : \mathsf{G} \parallel \mathcal{M}}$$

Figure 3.2: Network typing

- $\emptyset \upharpoonright^{\triangleright} (\mathsf{p}, \mathsf{q})$ is the sequence of labels occurring in messages from p to q in $\mathcal{M}$, defined by:

$$\emptyset \upharpoonright^{\triangleright} (\mathsf{p}, \mathsf{q}) = \epsilon \qquad (\langle \mathsf{r}, \lambda, \mathsf{s} \rangle \cdot \mathcal{M}) \upharpoonright^{\triangleright} (\mathsf{p}, \mathsf{q}) = \begin{cases} \lambda \cdot (\mathcal{M} \upharpoonright^{\triangleright} (\mathsf{p}, \mathsf{q})) & \text{if } \mathsf{r} = \mathsf{p} \text{ and } \mathsf{s} = \mathsf{q}, \\ \mathcal{M} \upharpoonright^{\triangleright} (\mathsf{p}, \mathsf{q}) & \text{otherwise.} \end{cases}$$

- the auxiliary judgment $\vdash_{\mathsf{read}} \langle \mathsf{G}, \mathsf{p}, \mathsf{q}, \zeta \rangle$ means that (each path of) G reads a sequence of messages from p to q with labels ζ. The straightforward (inductive) definition of this judgment is given in the bottom part of Fig. 3.1.

In rules (Out-Rcv) and (Out-Ext), the projection on another participant is only defined if projecting on such participant G_i, for $i \in I$, and the queue where the message has been added, gives processes P'_i possibly different, but which can be smoothly combined to provide the resulting projection. More precisely, the common structure of all such processes is modelled by a multi-hole context $\mathcal{C}$, where, for each hole $j \in J$, this is filled by a different process subterm in each P'_i. Such different process subterms start with an input from the same sender, and different labels[1]. In this way, they can be combined in an internal choice which is used to fill the context in the resulting projection.

The only difference between the two rules is that, in rule (Out-Rcv), the same sender and the different labels should be those of the output type to be projected, whereas in rule (Out-Ext) they are arbitrary.

Rules in Fig. 3.1 define a relation, while usually projection is expected to be a function. However, it can be proved that for fair global types this is the case.

Type checking The network type checking is formalized in Fig. 3.2. Type checking succeeds if two conditions hold:

- For each participant, the associated process in the network should be *consistent* with that obtained as projection of the global type.
- The set of participants of the global type is a subset of the participants of the network.

Moreover, recall that the global type is assumed to be fair.

In the first condition, the consistency relation expresses the fact that the *protocol* specified through the global type can be more general than the process in the network, and it is

[1] In rule (Out-Ext) this follows from the well-formedness of the process in the consequence.

$$(\leq\text{-OUT})\ \frac{\mathsf{P}_i \leq \mathsf{Q}_i \quad i \in I}{\mathsf{p}!\{\lambda_i.\mathsf{P}_i\}_{i\in I} \leq \mathsf{p}!\{\lambda_i.\mathsf{Q}_i\}_{i\in I\cup J}} \qquad (\leq\text{-IN})\ \frac{\mathsf{P}_i \leq \mathsf{Q}_i \quad i \in I}{\mathsf{p}?\{\lambda_i.\mathsf{P}_i\}_{i\in I\cup J} \leq \mathsf{p}?\{\lambda_i.\mathsf{Q}_i\}_{i\in I}} \qquad (\leq\text{-}\mathbf{0})\ \frac{}{\mathbf{0} \leq \mathbf{0}}$$

Figure 3.3: Preorder on processes

formalized as a preorder on processes. In the second condition, note that, whereas the global type obviously cannot mention participants not present in the network, there could be participants which are not mentioned in the global type. In this way it is guaranteed that, if a network is well-typed, then also the equivalent networks obtained adding an arbitrary number of inactive processes are well-typed.

The preorder on processes is defined in Fig. 3.3.

In rule (≤-Out), to be consistent with a protocol which is an internal choice, a process should be an internal choice with the same or less labels, and, for each of them, behave consistently with the protocol. Intuitively, the process may never send some label, and this can be seen as a more specific version of the protocol. A symmetric reasoning can be done for rule (≤-In). That is, to be consistent with a protocol which is an external choice, a process should be an external choice with at least those labels, and, for each of them, behaving consistently with the protocol. Intuitively, the process may wait for some label that will never be received; this again can be seen as a more specific version of the protocol. In rule (≤-$\mathbf{0}$), the inactive process is consistent with itself.

The properties enforced by the type system for a well-typed network are:

- a participant whose associated process starts with an internal choice will eventually find one the corresponding messages on the queue;
- a message on the queue will be eventually read.

The definition of the operational semantics and the proof of such properties is outside the aim of the current thesis, which focuses on providing a mechanical verification of the network typing rule.

Chapter 4

Implementation

In this chapter we describe how the formal definitions in Chapter 3 are implemented in co-logic programming. We only report the interesting portions of code. The complete code, together with a test suite, and instructions for using the prototype, can be found at `https://github.com/RiccardoBianc/Asynchronous-global-types-implementation`.

4.1 Processes, networks and global types

Processes are defined coinductively, so they are implemented by a coinductive predicate `process`. A predicate is declared to be coinductive by using the directive `coinductive`, followed by the name of the predicate and its arity, as shown below.

```
:- coinductive process/1.
```

The predicate is defined by three clauses, corresponding to the three productions in the definition given in Sect. 3.1.

```
process(send_process(A,[Lambda-P|LPs])) :-
        participant_name(A),
        map_from_to(label,process,[Lambda-P|LPs]),!.

process(receive_process(A,[Lambda-P|LPs])) :-
         participant_name(A),
         map_from_to(label,process,[Lambda-P|LPs]),!.

process(zero).
```

We use Prolog variables `A`, `B`, `C` for participants, `Lambda` for labels, and `P` for processes. The

notation `Lambda-P` is used in SWI-Prolog to denote a pair (called *key-value pair*), in this case consisting of a label `Lambda` and a process `P`. Finally, following a widely-used Prolog convention, we use `Xs` for a list of `X`, so the variable `LPs` denotes a list of pairs `Lambda-P`.

The predicate `participant_name` checks that the argument is a participant name (in the implementation, a string). The predicate `map_from_to`, with arguments `Dom`, `Cod`, and `KVs`, checks that the list of pairs `KVs` represents a map with domain `Dom` and codomain `Cod`. In the code above, it checks that it is a map from labels into processes, that is, keys are labels (in the implementation, strings), values are processes, and there are no repeated keys. The same predicate will be instantiated in the following to implement queues, networks, global types, and contexts. The definition is the following.

```
map_from_to(Dom,Cod,KVs) :-
 unique_keys(KVs),
 maplist(({Dom,Cod}/[K-V]>> call_pair(Dom,Cod,K-V)), KVs).

unique_keys(KVs) :- pairs_keys(KVs,Ks), is_set(Ks).

call_pair(X,Y,Left-Right):-  call(X,Left), call(Y,Right).
```

The predicate `unique_keys` extracts the keys from a list of pairs and checks that the resulting list is a set. Then, it is checked that, for each pair `K-V` in `KVs`, `K` satisfies predicate `Dom`, and `V` satisfies predicate `Cod`. This is obtained by using the higher-order features offered by SWI-Prolog. Notably, the predefined higher-order predicate `maplist` applies a goal (predicate) to all the elements of a list, and in the code above this goal is a lambda expression. The parameters in curly brackets (`Dom` and `Cod`) are shared with the surrounding context, so a goal with only one argument `K-V` is obtained. The definition of `call_pair` uses the predefined higher-order predicate `call`, whose first argument is a goal (predicate), which is called on the second argument.

Here and in the following, code contains *cuts* `!`, which are mostly inserted only for optimization purpose.

As described in Sect. 2.2, regular processes can be represented by equations. For instance, the following process, written using the syntax of Sect. 3.1:

$\mathsf{P} = \mathsf{p}!\lambda.\mathsf{q}?\lambda.\mathsf{P}$

can be represented as follows:

```
P = send_process("p",["lambda"-receive_process("q"["lambda"-P])])
```

The message queue is defined in Sect. 3.1 as a list of messages, that is, triples $\langle \mathsf{p}, \lambda, \mathsf{q} \rangle$, modulo the equivalence that the only order that matters is that between messages with the same sender and receiver. As mentioned there, this structural equivalence amounts to say that a message queue can be seen as a map from pairs p, q of participants to sub-queues

which are lists of labels. The implementation follows the latter approach, which makes easy to find the right sub-queue maintaining the order.

```
label_list([]).
label_list(Lambdas) :-
        maplist(label,Lambdas).

participant_pair(A-B) :-
        participant(A),
        participant(B).

queue([]).

queue([A-B-Lambdas|M]) :-
        map_from_to(participant_pair,label_list,[A-B-Lambdas|M]).
```

As for other maps, the message queue `M` is implemented as a list of pairs; the first element (key) of the pair is in turn a pair, consisting of a sender `A` and a receiver `B`, the second element (value) is the list `Lambdas` of labels sent from `A` to `B`. The predicate `map_from_to`, described above, checks that this list of pairs represents a map from pairs of participants into lists of labels.

Networks are implemented by the following predicate:

```
network([A1-P1,A2-P2|APs]-M) :-
        map_from_to(participant,process,[A1-P1,A2-P2|APs])
        queue(M).
```

A network is a pair consisting of a list with at least two elements, and a message queue. The elements of the list are pairs participant-process. As in processes and queues above, it is checked that this list of pairs represents a map from participants to processes.

As processes, global types are defined coinductively, so they are implemented by a coinductive predicate `global_type`. The predicate is defined by three clauses, corresponding to the three productions in the definition given in Sect. 3.2.

The functor[1] `output_type` has three arguments: the sender, the receiver, and a (non-empty) list of pairs consisting of a label and a global type. The functor `input_type` has four arguments: the sender, the receiver, the label, and the following global type. Finally, the functor `end` implements the inactive global type.

```
global_type(output_type(A,B,[Lambda-G|LGs])) :-
        A \= B,
        participant_name(A),
```

[1]That is, function symbol.

```
        participant_name(B),
        map_from_to(label,global_type,[Lambda-G|LGs]).

global_type(input_type(A,B,L,G)) :-
        A \= B,
        participant_name(A),
        participant_name(B),
        label(L),
        global_type(G).

global_type(end).
```

The predicate `map_from_to` applies the predicates `label` and `global_type` to all pairs in the list, as explained before.

Participants of a global type are computed as follows.

```
participants(output_type(A,B,[Lambda-G|LGs]),Res) :-
        participants_list([Lambda-G|LGs],As),
        union([A],As,As_whith_A),
        union([B],As_whith_A,As_with_A_and_B),
        permutation(As_with_A_and_B,Res).

participants(input_type(A,B,_,G),Res) :-
        participants(G,As),
        union([A],As,As_whith_A),
        union([B],As_whith_A,As_with_A_and_B),
        permutation(As_with_A_and_B,Res).

participants(end,[]).

participants(end,[_|_]).

participants_list([_-G],As) :-
        participants(G,As).

participants_list([_-G|LGs],As3) :-
        participants(G,As1),
        participants_list(LGs,As2),
        union(As1,As2,As3).
```

The predicate is declared coinductive. The global type is scanned recursively checking that the participants present at each step are present in the result set, so that repetitions are removed and the order is immaterial. The predicate `participants_list` applies to the

second component of each pair in the list of pairs the predicate `participants`. When a global type is encountered twice, the built-in cycle detection of Prolog accepts the goal. Note that the predicate as defined above also holds when the set of participants of the global type is a *subset* of the second argument. To implement a version of predicate in which the two sets should be equal, we should use a by-hand cycle detection mechanism. That is, when the same global type is encountered a second time, the predicate shoud hold only if it is associated with the same set of participants as the first time. However, in our implementation this predicate is only used for the typing predicate, where it is enough to check subset inclusion, see Sect. 4.3.

The implementation of the notion of player shows an interesting use of coinductive features. On an infinite (regular) global type, an inductive definition of `player` would not terminate in the negative case (an argument which is not a player), while a coinductive definition would be not correct, since it would be successful, when finding a cycle (that is, the same global type), for an arbitrary argument. The solution is to define `player` as the negation of a coinductive predicate `not_player`, as shown below.

```
player(G,A) :- \+not_player(G,A).

not_player(output_type(A,_,LGs), B):-
        B \= A,
        not_player_list(LGs, B).

not_player(input_type(_,B,_,G), A) :-
        A \= B,
        not_player(G,A).

not_player(end,_).
```

Being this predicate coinductive, when a cycle is found the call succeeds, hence the predicate `player` fails, correctly, in the negative case. On the other hand, in the positive case (an argument which is a player) the predicate `not_player` finitely fails, hence the predicate `player` succeeds.

In Sect. 3.2 at page 32, *fairness* of a global type (a participant cannot wait forever) is expressed by requiring that, for all participants `p` and subterm `G` of the global type, the depth of `p` in `G` is finite. This is implemented by the predicate `fair` below.

```
fair(G) :-
        participants(G,As),
        fair_list(G,As).

fair_list(_,[]).
```

```
fair_list(G,[A|As]) :-
        all_finite_depth(G,A), fair_list(G,As).
```

The set of participants of the global type is computed, and then, for each participant, it is checked that its depth in each subterm of the global type is finite, by the predicate `all_finite_depth`, described below.

The depth of a participant p in G is defined in Sect. 3.2 in an abstract way, by computing its depth in each of the paths of G. In particular, if p is a player of G, but it does not occur as player in some path of G, then $\mathsf{depth}(\mathsf{G},\mathsf{p}) = \infty$, modelling the fact that p may wait forever.

To enforce fairness in an algorithmic way, in the implementation we take a different approach, by defining the predicate `finite_depth` which holds if a participant has finite depth in a global type. That is, if the participant is a player, then it occurs as player in each path of the given global type. Then, we define the predicate `all_finite_depth` which checks that `finite_depth` holds for each subterm of the given type.

```
finite_depth(G,A,_) :-
        not_player(G,A).

finite_depth(output_type(A,_,_),A,_).

finite_depth(input_type(_,A,_,_),A,_).

finite_depth(output_type(A,B,LGs),C,G_found) :-
        \+member(output_type(A,B,LGs),G_found),
        finite_depth_list(LGs,C,[output_type(A,B,LGs)|G_found]).

finite_depth(input_type(A,B,_,G),C,G_found) :-
        \+member(input_type(A,B,_,G),G_found),
        finite_depth(G,C,[input_type(A,B,_,G)|G_found]).
```

In the first clause, if the participant is not a player of the global type, then the depth is 0, so it is finite.

In the second and third clause, if the participant is a player in the root node, then the depth is 1, so it is finite. Otherwise, we have to check that the participant is a player for all the paths starting from the children nodes. To avoid non-termination in this check, we use a by-hand cycle detection mechanism, implemented with the argument `G_found`. This argument is the list of already encountered global types, which grows at each recursive call. When the same global type is encountered twice, that i, is already in `G_found`, then the goal is rejected, because it means that following that path the participant has not been found as a player, so its depth is infinite.

The predicate `finite_depth_list`, not reported, is the lifting to lists of pairs label-global type of the predicate.

Finally, note that there is no clause for the inactive process because this case is covered by the first clause.

The above predicate is applied to all the sub-terms of a global type by the predicate `all_finite_depth`.

```
all_finite_depth(output_type(A,B,LGs),C) :-
        finite_depth_list(output_type(A,B,LGs),C,[]),
        all_finite_depth_list(LGs,C).

all_finite_depth(input_type(A,B,Lambda,G),C) :-
        finite_depth_list(input_type(A,B,Lambda,G),C,[]),
        all_finite_depth(G,C).

all_finite_depth(end,_).
```

This predicate is declared coinductive, because in this case when the goal is encountered twice it must be accepted. The predicate `all_finite_depth_list`, not reported, is the lifting to lists of pairs label-global type of the predicate.

4.2 Projection

The projection definition uses process contexts with an arbitrary number of holes indexed with natural numbers, where we assume that each hole has a different index. Hence, first of all we describe the predicate `context`, implementing such contexts.

```
context(hole).

context(Ctx) :-
        process(Ctx).

context(send_process(A,[Lambda-Ctx|LCtxs])) :-
        participant_name(A),
        map_from_to(label,context,[Lambda-Ctx|LCtxs]).

context(receive_process(A,[Lambda-Ctx|LCtxs])) :-
        participant_name(A),
        map_from_to(label,context,[Lambda-Ctx|LCtxs]).
```

The predicate is defined by four clauses, corresponding to the four productions in the

definition given in Sect. 3.3. In the last two clauses contexts are implemented by the same functors as for processes. However, in this case the base cases are either a hole (first clause) or a process (second clause). Note that the predicate is defined inductively, since the context should contain an arbitrary, but finite, number of holes. In other words, a context term can be infinite only if it contains an infinite process subterm.

Recall that, given a context $\mathcal{C}$ with holes indexed in J, we denote by $\mathcal{C}[\mathsf{P}_j]_{j\in J}$ the process obtained by filling the hole indexed by j by P_j, for all $j \in J$. This operation is implemented by the predicate `fill_context`, which has three arguments: the context, the list of processes replacing the holes, and the resulting process. The predicate visits the context following a DFS approach and, when a hole is found, it is replaced it with the first process in the list. Processes are expected to be exactly as many as the holes, otherwise the predicate fails, as detailed below.

```
fill_context(Ctx,Ps,P) :- fill_aux(Ctx,Ps,P,[]),!.

fill_aux(hole,[P|Ps],P,Ps) :- !.

fill_aux(Ctx,Ps,Ctx,Ps) :- process(Ctx),!.

fill_aux(send_process(A,LCtxs),Ps,send_process(A,LPs),Remaining) :-
        fill_aux_list(LCtxs,Ps,LPs,Remaining),!.

fill_aux(receive_process(A,LCtxs),Ps,
                receive_process(A,LPs),Remaining) :-
        fill_aux_list(LCtxs,Ps,LPs,Remaining),!.
```

The auxiliary predicate `fill_aux` has four arguments: the first three arguments are the same as before, the fourth argument is the part of the list of processes which is not consumed yet. To force to consume all the processes, the main predicate sets to the empty list this argument. The first clause returns the first process in the list when the context is a hole, hence the predicate fails when the processes are less than the holes. The second clause returns the context itself if it is a process, so this is the case of a context with no holes. The remaining two clauses are the propagation of the predicate to the subterms, implemented by the predicate `fill_aux_list`.

The projection is implemented by the predicate `projection`, which takes four arguments: the global type, the message queue, the target participant, and the process resulting as projection.

```
projection(G,M,A,P) :-
        list_to_assoc(M,M_assoc),
        empty_assoc(GPM_found),
        projection_cycle_detect(GPM_found,G,M_assoc,A,P).
```

The predefined predicate `list_to_assoc` converts a list of key-value pairs (as the queue `M` is implemented, see before) into an *association list*, an SWI-Prolog structure which has more efficient operations. The auxiliary predicate `projection_cycle_detect` requires one more argument, the map `GPM_found`, which keeps trace of already encountered global types, as motivated and detailed below.

This additional argument is necessary to guarantee the termination of the predicate, which in this case would be not ensured by the built-in mechanism of cycle detection for coinductive predicates. Indeed, this mechanism detects a cycle when the same goal is found, whereas in this case it should be detected when the same global type is found. In fact, it is possibile that the same global type is found, but the queue has grown.

To solve this problem, we defined the predicate as inductive and implemented by hand the cycle detection, adding an argument `GPM_found` which is a map from global types to pairs process-queue. In this way, when a global type is encountered twice, we check that it is associated with the same process and the same queue. This map is updated every time a new global type is found, thus avoiding non termination, since global types are regular terms.

The predicate `projection_cycle_detect` will be described clause by clause below. The following auxiliary predicate is used, which adds a global type `G`, with the associated process `P` and queue `M`, to `GPM_found`, if not present yet.

```
add_if_not_present(GPM_found,G,P,M,GPM_found_modified) :-
  assoc_to_keys(GPM_found,Gs_found),
  \+member(G,Gs_found),
  put_assoc(G,GPM_found,P-M,GPM_found_modified).
```

The predefined predicate `assoc_to_keys` extracts from the map the list of already encountered global types, and `put_assoc` updates `GPM_found` returning`GPM_found_modified`.

The first clause handles the case of an already found global type:

```
projection_cycle_detect(GPM_found,G,M,_,P) :-
  get_assoc(G,GPM_found,P-M),!.
```

Rule (Out-Rcv) is modelled by two clauses. Recall that in this rule the projection of an output type $\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}$ is only defined if projecting each G_i gives processes P_i with a common structure, modelled by a multi-hole context $\mathcal{C}$, and, moreover, the "fillings", that is, the process subterms replacing the holes, for process P_i, all start with an input from p, and label λ_i. That is, $\mathsf{P}_i = \mathcal{C}[\mathsf{p}?\lambda_i.\mathsf{P}_{i,j}]_{j\in J}$. In this way, they can be combined in an external choice which is used to fill the context in the resulting projection.

```
projection_cycle_detect(GPM_found,
                  output_type(A,B,[Lambda-G]),M,C,P) :-
  C \= A,
```

```
  add_if_not_present(GPM_found,output_type(A,B,[Lambda-G]),P,M,
                GPM_found_modified),
  add_to_queue(A,B,Lambda,M,M_modified),!,
  projection_cycle_detect(GPM_found_modified,G,M_modified,B,P),!.

projection_cycle_detect(GPM_found,
                output_type(A,B,[Lambda1-G1,Lambda2-G2|LGs]),
                M,B,P) :-
  add_if_not_present(GPM_found,
                output_type(A,B,[Lambda1-G1,Lambda2-G2|LGs]),
                P,M,GPM_found_modified),
  projection_list(GPM_found_modified,A,B,M,
                [Lambda1-G1,Lambda2-G2|LGs],B,
                [Lambda1-P1,Lambda2-P2|LPs]),!,
  build_context(P1,P2,A,Lambda1,Lambda2,Context),
  pairs_keys(LGs,Lambdas),
  pairs_values(LPs,Ps),
  check_each_process(Context,A,[Lambda1,Lambda2|Lambdas],
                [P1,P2|Ps],LPss),
  build_process_result(Context,LPss,A,P).
```

The first clause corresponds to an output type with only one pair `Lambda-G`, the second clause to an output type with more than one such pair.

In the first case, it is enough to project the unique subterm `G`. Otherwise, the predicate `projection_list` projects all the subterms, obtaining a list of processes, which should be checked to have a common structure in the sense explained above.

The predicate `build_context` scans the first two processes `P1` and `P2`, checking they are equal apart from subterms which start with an input from the sender `A`, and labels `Lambda1` and `Lambda2`, respectively. In this case, the variable `Context` will contain the context corresponding to the common part among the two processes, with holes in place of their different subterms.

After extracting the remaining labels `Lambdas` of the output type, and the remaining processes `Ps` obtained as projections, the predicate `check_each_process`, reported in the following, checks that all the processes can be obtained by filling the holes of `Context`, and, moreover, that the fillings for the i-th process all start with an input from `P`, and the i-th label. That is, in the notation of the rule, $\mathsf{P}_i = \mathcal{C}[\mathsf{p}?\lambda_i.\mathsf{P}_{i,j}]_{j \in J}$.

If it is the case, then the result `LPss` is a list with an element for each hole, where the element for the j-th hole is a list of pairs, one for each process, where the pair for the i-th process is (the representation of) $\langle \lambda_i, \mathsf{P}_{i,j} \rangle$. Finally, the predicate `build_process_result`, not reported, builds the result, which is the process `P` obtaining by filling each hole of

`Context`, notably, the j-th hole by the external choice $\mathsf{p}?\{\lambda_i.\mathsf{P}_{i,j}\}_{i\in I}$.

Also rule (Out-Ext) is modelled by two clauses, where the first one is the same described for (Out-Rcv) above.

```
projection_cycle_detect(GPM_found,output_type(A,B,
                 [Lambda1-G1,Lambda2-G2|LGs]),M,
                 C, P):-
  A\=C, B\=C,
  add_if_not_present(GPM_found,
                 output_type(A,B,[Lambda1-G1,Lambda2-G2|LGs]),
                 P,M,GPM_found_modified),
  player(output_type(A,B,[Lambda1-G1,Lambda2-G2|LGs]),C),
  projection_list(GPM_found_modified,A,B,M,
                 [Lambda1-G1,Lambda2-G2|LGs],C,
                 [Lambda1-P1,Lambda2-P2|LPs]),!,
  build_context(P1,P2,R,Lambda1_prime,Lambda2_prime,Context),
  pairs_values(LPs,Ps),
  check_each_process(Context,R,[Lambda1_prime,Lambda2_prime|_],
                 [P1,P2|Ps],LPss),
  build_process_result(Context,LPss,R,P).
```

The second clause is analogous to the second clause for (Out-Rcv), with the only difference that in this case the predicate `build_context` is called with variables `R`, `Lambda1_prime`, and `Lambda2_prime` which are free, corresponding to the fact that the rule can be instantiated with arbitrary sender and labels. However, the sender must be the same for all processes, and each label should be the same for all the fillings for a process.

Rule (Out-Snd) is modelled by the clause below.

```
projection_cycle_detect(GPM_found,output_type(A,B,LGs),M, A,
    send_process(B,P_children)) :-
  add_if_not_present(GPM_found,output_type(A,B,LGs),
    send_process(B,P_children),M,GPM_found_modified),
    consume_queue_list(LGs,A,B,M),
  projection_list(GPM_found_modified,A,B,M,LGs,A,P_children),!.
```

The clause, after updating the cycle detection map, checks with `consume_queue_list` that each global type in the list `LGs` will eventually read the corresponding sent message. That is, this predicate, reported in the following, implements the side condition $\vdash_{\mathsf{read}} \langle \mathsf{G}_i, \mathsf{p}, \mathsf{q}, \mathcal{M}\!\upharpoonright\!(\mathsf{p},\mathsf{q}) \cdot \lambda_i\rangle\ \forall i \in I$. After this check, the clause propagates the projection to the subterms.

The rule (In-Rcv) is modelled by the clause below.

```
projection_cycle_detect(GPM_found,input_type(A,B,Lambda,G_children),
```

```
    M,B,receive_process(A,[Lambda-P_children])) :-
  add_if_not_present(GPM_found,input_type(A,B,Lambda,G_children),
    receive_process(A,[Lambda-P_children]),M,GPM_found_modified),
  remove_from_queue(M,A,B,Lambda,M1),!,
  projection_cycle_detect(GPM_found_modified,G_children,
    M1,B,P_children),!.
```

The clause removes the received message from the queue, using the last argument of predicate `remove_from_queue` as output, and then propagates the projection to the subterms with the new queue.

The rule (In-Ext) is expressed by the clause below.

```
projection_cycle_detect(GPM_found,input_type(A,B,Lambda,Gs),M,C,P) :-
        C\=B,!,
        add_if_not_present(GPM_found,input_type(A,B,Lambda,Gs),P,M,
                GPM_found_modified),
        player(input_type(A,B,Lambda,Gs),C),
        remove_from_queue(M,A,B,Lambda,M1),!,
        projection_cycle_detect(GPM_found_modified,Gs,M1,C,P).
```

The clause checks the side conditions that `C` differs from the receiver `B`, and that `C` is a player in the global type. Then, it removes from the queue the received message, and it propagates the projection to the following global type with the new queue.

We provide now the implementation of some predicates informally described above. The predicate `build_context` has six arguments: two processes, a participant `A`, two labels `Lambda1`, `Lambda2`, and a context. c ho eliminato gli ultimi due argomenti e scambiato le due regole centrali c The predicate checks that the two processes are either equal, or differ starting from input nodes from participant `A`, where the latter and former process receive labels `Lambda1` and `Lambda2`, respectively.

```
build_context(zero,zero,_,_,_,zero).

build_context(receive_process(A,[Lambda1-P1]),
    receive_process(A,[Lambda2-P2]), A,Lambda1,Lambda2,
    hole) :-
  Lambda1 \= Lambda2.

build_context(receive_process(B,LP1),receive_process(B,LP2),A,Lambda1,
    Lambda2,receive_process(B,Context)) :-
  build_context_list(LP1,LP2,A,Lambda1,Lambda2,Context).

build_context(send_process(B,LP1),send_process(B,LP2),A,Lambda1,
    Lambda2,send_process(B,Context)) :-
```

```
build_context_list(LP1,LP2,A,Lambda1,Lambda2,Context).
```

The predicate is declared coinductive, because if the two processes are regular infinite terms the predicate must hold if no difference is found. The predicate scans the processes recursively checking at each step their shape. The second clause checks that the labels of the two terms are different. In this case, the scan stops, and a hole is returned as context, otherwise the scan should continue.

The predicate `check_each_process` has six arguments: a context, a participant `P`, a list of labels, a list of processes, and a result `LPss`. As already mentioned, the predicate checks that all the processes can be obtained by filling the holes of `Context`, and, moreover, that the fillings for the i-th process all start with an input from `P`, and the i-th label. If it is the case, then the result `LPss` is a list with an element for each hole, where the element for the j-th hole is a list of pairs, one for each process, where the pair is the i-th label and the filling for that hole.

```
check_each_process(Context,_,[_],[P],[]) :-
        process(Context),
        fill_context(Context,[],P).

check_each_process(Context,A,[_|Lambdas],[P|Ps],[]) :-
        process(Context),
        fill_context(Context,[],P),
        check_each_process(Context,A,Lambdas,Ps,_).

check_each_process(Context,A,[Lambda],[P],LPss) :-
        fill_context(Context,Fillings,P),
        check_branch(Fillings,A,Lambda,LPs).
        cons_branch(LPs,LPss).

check_each_process(Context,A,[Lambda|Lambdas],[P|Ps],
                LPss) :-
        fill_context(Context,Fillings,P),
        check_branch(Fillings,A,Lambda,LPs),
        check_each_process(Context,A,Lambdas,Ps,LPss_Tail),
        \+member(Lambda,Lambdas),
        add_branch(LPs,LPss_Tail,LPss).
```

The first two clauses handle the case when the context is a process (that is, has no holes). In this case, it is enough to check that each process in the list is equal to `Context`, and this is obtained calling the `fill_context` predicate with the empty list of fillings. The first and second clause deal with a list of processes with only one, and more than one, element, respectively.

Otherwise, the following two rules are applied, which again deal with a list of processes with only one, and more than one, element, respectively.

For each process in the list, the predicate `fill_context` finds the corresponding fillings (process subterms) to be inserted in place of the holes. As said above, the fillings for the i-th process should all be of shape $\mathsf{p}?\lambda_i.\mathsf{P}_{i,j}$. This is checked by the predicate `check_branch`, not reported, which also transforms each such filling into a pair $\langle \lambda_i, \mathsf{P}_{i,j} \rangle$, and returns (the representation of) such list of pairs in `LPs`.

After that, if the list of processes has only one element, then the list of pairs `LPs` is transformed by the predicate `cons_branch`, not reported, in a list of list of pairs where each element has length one. If the list of processes has more than one element, then `check_each_process` is recursively called, returning `LPss_Tail` as result, which is completed with the current result by the predicate `add_branch`, not reported.

The predicate `consume_queue_list` has four arguments: a list of pairs label-global type, a sender, a receiver, and a list of labels. As already said, it implements the side condition $\vdash_{\mathsf{read}} \langle \mathsf{G}_i, \mathsf{p}, \mathsf{q}, \mathcal{M}\!\upharpoonright\!(\mathsf{p},\mathsf{q}) \cdot \lambda_i \rangle \ \forall i \in I$ of rule (Out-Snd).

```
consume_queue_list([Lambda-G],A,B,M) :-
        add_to_queue(A,B,Lambda,M,M1),
        get_assoc(A-B,M1,Lambdas),
        consume_queue(G,A,B,Lambdas).

consume_queue_list([Lambda-G|LGs],A,B,M) :-
        add_to_queue(A,B,Lambda,M,M1),
        get_assoc(A-B,M1,Lambdas),
        consume_queue(G,A,B,Lambdas),
        consume_queue_list(LGs,A,B,M).

consume_queue(G,A,B,Lambdas) :-
  \+not_consume_queue(G,A,B,Lambdas).

not_consume_queue(end,_,_,[_|_]).

not_consume_queue(input_type(A,B,Lambda,G),A,B,[Lambda|Zeta]) :-
        not_consume_queue(G,A,B,Zeta).

not_consume_queue(input_type(A,B,Lambda1,_),A,B,[Lambda2|_]) :-
        Lambda1 \= Lambda2.

not_consume_queue(input_type(A,B,_,G),C,D,Zeta) :-
        (A \= C ; B \= D),
        not_consume_queue(G,C,D,Zeta).
```

```
not_consume_queue(output_type(_,_,LGs),A,B,Zeta) :-
        not_consume_queue_children(LGs,A,B,Zeta),!.
```

The predicate `consume_queue` implements the judgment $\vdash_{\mathsf{read}} \langle \mathsf{G}, \mathsf{p}, \mathsf{q}, \zeta \rangle$ defined in the bottom part of Fig. 3.1. This judgment cannot be implemented inductively, since it would not terminate when the predicate does not hold. The solution is to implement the negation of the judgment as a coinductive predicate. The predicate `not_consume_queue` has four arguments: a global type, a sender, a receiver, and a list of labels. The predicate holds if the global type does *not* consume all the labels. This can happen for two reasons: either, after removing the labels from the queue for each input node with the given participants, the queue is not empty, or the order of labels in the queue is not the same order of input labels. The negation of the predicate corresponds to the judgment $\vdash_{read}$ as expected, that is, a predicate that holds if all the labels will be eventually consumed.

4.3 Type checking

The following predicate implements typing rule (T-Net).

```
typing([A-P,B-Q|APs]-M,G-M) :-
        project_net([A-P,B-Q|APs],G,M),
        pairs_keys([A-P,B-Q|APs],As),
        participants(G,As).

project_net([A-P],G,M) :-
        projection(G,M,A,P_first),
        process_preorder(P,P_first).

project_net([A-P|APs],G,M) :-
        projection(G,M,A,P_first),
        process_preorder(P,P_first),
        project_net(APs,G,M).
```

The body of the clause for `typing` checks the premises of the rule. The first premise is checked by `project_net` . That is, for each participant of the network, the projection is computed, and the preorder constraint is checked on the result. Predicate `pairs_keys` extracts the keys from the list of pairs, to obtain the participants of the network, then, using the predefined predicate `subset`, it is checked that the participants of the global type, obtained with `participants`, are a subset of the participants of the network. Finally, the global type is checked to be fair (condition implicitly assumed in the rule).

The following predicate implements the preorder on processes. The three clauses correspond to rules ($\leq$-Out), ($\leq$-In), and r($\leq$-**0**), espectively.

```
process_preorder(send_process(A,[Lambda-P|LPs]),
        send_process(A,[Lambda-Q|LQs])) :-
  pairs_keys([Lambda-P|LPs],L1s),
  pairs_keys([Lambda-Q|LQs],L2s),
  subset(L2s,L1s),
  process_preorder_list([Lambda-P|LPs],[Lambda-Q|LQs],L1s).

process_preorder(receive_process(A,[Lambda-P|LPs]),
        receive_process(A,[Lambda-Q|LQs])) :-
  pairs_keys([Lambda-P|LPs],L1s),
  pairs_keys([Lambda-Q|LQs],L2s),
  subset(L1s,L2s),
  process_preorder_list([Lambda-P|LPs],[Lambda-Q|LQs],L2s).

process_preorder(zero,zero).
```

To check the premises, we have to check two costraints: the two set of labels are in a subset relation, and the processes associated with the same label are, recursively, in the preorder relation. To verify the costraints, first the predicate `pairs_keys` extracts from the list of pairs of both arguments the labels, and then the predefined predicate `subset` checks they are in the subset relation. The recursive check is performed by the predicate `process_preorder_list`.

```
process_preorder_list([Lambda-P],LQs,Lambdas) :-
        member(Lambda,Lambdas),
        pairs_keys_values(LQs,[Lambda],[Q]),
        process_preorder(P,Q).

process_preorder_list([Lambda-_],_,Lambdas) :-
        \+member(Lambda,Lambdas).

process_preorder_list([Lambda-_|LPs],LQs,Lambdas) :-
        \+member(Lambda,Lambdas),
        process_preorder_list(LPs,LQs,_).

process_preorder_list([Lambda-P|LPs],LQs,Lambdas) :-
        member(Lambda,Lambdas),
        pairs_keys_values(LQs,[Lambda],[Q]),
        process_preorder(P,Q),
        process_preorder_list(LPs,LQs,Index).
```

This predicate has three arguments: the list of pairs label-process of the right argument, the list of pairs label-process of the left argument, and a set of labels. This set is different depending on the rule: when used by (≤-In), it is the set of labels of the right argument, whereas when used by rule (≤-Out) it is the set of labels of the left argument. The predicate scans the list of the left argument, checking whether the current label is present in `Lambdas`, and then appling recursively the predicates `process_preorder` and `process_preorder_list`. The membership test is necessary, since in rule (≤-In) we do not consider labels that belong only to the left argument. The second and third clauses ignore the labels continuing to scan if the list has at least two elements or stopping the scan otherwise. In rule (≤-Out), while scanning we are sure that the left labels are all present in the right arguments, so this check is useless, but we preferred to implement only one predicate, avoiding repetion of similar code.

Chapter 5

Conclusion

The outcome of the thesis is an implementation in co-logic programming of a novel formulation of global types for asynchronous netwoks, where asynchrony is expressed at the level of the type system.

In the first two chapters we have described the state-of-the-art for the thesis work. Notably, we have illustrated the classical notion of global types proposed in literature, based on three abstraction levels of description of the the network: the global level, the session level and the process level. The notion of projection provides a link between global and session level, whereas type checking provides a link between session level and process level. The aim of such formalisms is to ensure relevant properties of networks. Moreover, we have introduced the fundamental ingredients of the coinductive approach used in the thesis work: coinductive definitions, as opposed to inductive definition, in the framework of inference systems, and co-logic programming, an extension of logic programming where predicates can be marked as coinductive.

After these background chapters, we have presented a novel formulation of global types, where infinite processes and global types are defined coinductively, rather than by an explicit fixed point operator, and, more importantly, global types directly model asynchrony. Finally, we have described in detail the implementation of this model in co-logic programming, notably motivating our choices to solve termination problems, frequently encountered in the development. Our solutions are based on coinductive techniques mixed with inductive ones, possibly also employing user-defined cycle detection mechamisms.

The thesis work can be continued in many directions. First of all, the current implementation is a prototype, which should be refined and equipped with a suitable user interface to become more usable.

On the theoretical side, we mentioned that standard global types introduce a subtyping

relation, at the level of local types, that is, types for a single participant, that in our setting are just processes, which allows to *anticipate* output operations [MYH09b, CDCY14]. The intuition behind this approach is that, in an asynchronous communication model, output operations are not blocking, hence anticipating them will not cause deadlocks. This approach has a big issue: the subtyping relation has been proved to be undecidable, hence it cannot be used in practice, as typechecking must always terminate [BCZ17b]. However, it provides a perfect reference model.

Therefore, we plan to compare on a formal basis our asynchronous global types with such a reference model (standard global types and synchronous sybtyping). This could be done, for instance, by showing that a process P obtained projecting an asynchronous global type G has a super-process Q, w.r.t. the asynchronous subtyping relation, which is the projection of a standard global type. To reach such a result, it could be useful to introduce a subtyping relation between asynchronous global types, which simulates the one between processes.

The other direction, namely, a process typable by a standard global type using asynchronous subtyping is typable also by an asynchronous global type, should not hold in general, as we conjecture our system is decidable. However, it would be interesting to investigate whether, relaxing a bit the conditions on asynchronous global types, it is possible to get also this direction, because, in this case, we could exploit our decidable system to characterize a decidable fragment of asynchronous subtyping.

Another objective we would like to pursue is on the methodological side. We plan to investigate the possibility of formalizing our global types in a proof assistant supporting coinduction, such as Agda or Coq. This would provide a stronger support to our results, as proofs, which in this context could be quite involved and error prone, would be automatically checked and certified. Furthermore, proof assistants could help us developing a certified type checker for asynchronous global types.

Bibliography

[AD15] Davide Ancona and Agostino Dovier. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*, 140(3-4):221–246, 2015.

[BCD+08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *Concurrency Theory - CONCUR'08*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. doi:10.1007/978-3-540-85361-9_33.

[BCZ17a] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/j.ic.2017.07.010.

[BCZ17b] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/j.ic.2017.07.010.

[CDCY14] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP'14*, pages 135–146. ACM Press, 2014. URL: http://www.di.unito.it/~dezani/papers/cdy14.pdf.

[CDPY15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming - SFM 2015*, volume 9104 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2015. doi:10.1007/978-3-319-18941-3_4.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *European Symposium on Programming - ESOP' 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM Press, 2008. doi:10.1145/1328897.1328472.

[LG09] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LY17] Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

[Mil99] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[MYH09a] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 316–332, Heidelberg, 2009. Springer.

[MYH09b] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[SBMG07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming - ICALP 2007*, volume 4596 of *Lecture*

Notes in Computer Science, pages 472–483. Springer, 2007. doi:10.1007/978-3-540-73420-8_42.

[SMBG06a] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.

[SMBG06b] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *International Conference on Logic Programming*, pages 330–345. Springer Berlin Heidelberg, 2006.