Lightening Global Types *

Tzu-Chun Chen

Università di Torino, Dipartimento di Informatica, Corso Svizzera 185, 10149 Torino, Italy

Abstract

Global session types prevent participants from waiting for never coming messages. Some interactions take place just for the purpose of informing receivers that some message will never arrive or the session is terminated. By decomposing a big global type into several light global types, one can avoid such kind of redundant interactions. This work proposes a framework which allows to easily decompose global types into light global types, preserving the interaction sequences of the original ones but for redundant interactions.

Keywords: Communication Centred Programming, Session Types, Global Types

1. Introduction

Since cooperating tasks and sharing resources through communications under network infrastructures (e.g. clouds, large-scale distributed systems, etc.) have become the norm and the services for communications are growing with increasing users, it is a need to give programmers an easy and powerful programming language for developing interaction-based software applications. For this aim, Scribble [2], a communication-based programming language built upon the theory of global types [3, 4], is introduced. A developer can use Scribble as a tool to code a global protocol, which stipulates any local endpoints (i.e. local applications) participating in it. The merits of global types, which describe global protocols, are (1) giving all local participants a clear map of what events they are involved in and what are the behaviours for those events, and (2) efficiently exchanging, sharing, and maintaining communication plans across platforms.

1.1. Motivation

However, global types do not ensure an efficient communication programming. The scenario of a global communication can be very complex, so it

*This work is a revised and extended version of [1], prepresented in the Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES). This work has been sponsored by Torino University/Compagnia San Paolo Project SALT.

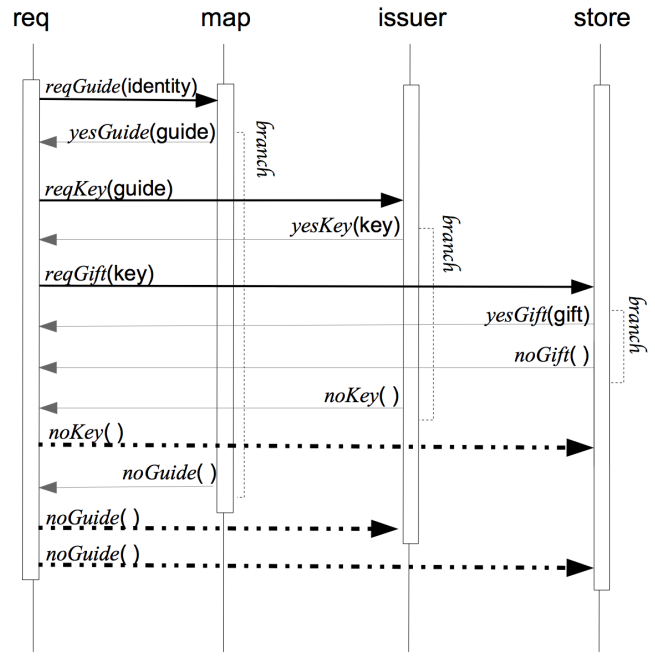

Figure 1: Nested Communications among `req` (i.e. the requester), `map`, `issuer`, and `store`.

becomes a burden for programmers to code interactions to satisfy protocols. At runtime, the cost for keeping all resources ready for a long communication and for maintaining the safety of the overall system can increase a lot.

Consider the global scenario, shown in Figure 1, which describes how a gift requester can get a *key* (with her identity) to fetch a wanted gift. Assume identity, guide, key, and gift are abstract type names, which can be types of string (denoted by Str), integer (denoted by Int), or bool (denoted by Bool). In order to get the key, she needs to get a *guide* from `map` for finding the key (communication labelled *reqGuide*). If the requester successfully gets the *guide*, she then proceeds to ask `issuer` for the *key* (communication labelled *reqKey*), and she can ask `store` for the gift with the given key (communication labelled *reqGift*); otherwise, she does not have the right to ask for it. First, it is not efficient nor economic to involve `issuer` and `store` at the stage while the requester negotiates with `map` (i.e. the guide provider) because `issuer` and `store` have nothing to do at this phase. Secondly, it will be no need to invoke `issuer` and `store` if the requester fails to get the *guide* (i.e. `map` replies `req` with *noGuide*()).

The dashed arrows are needed only because both `issuer` and `store` are invoked when the communication starts. They are used to inform `issuer` and `store` to terminate. If we do not invoke `issuer` and `store` at the beginning, these interactions represented by dashed lines are *redundant* : they become

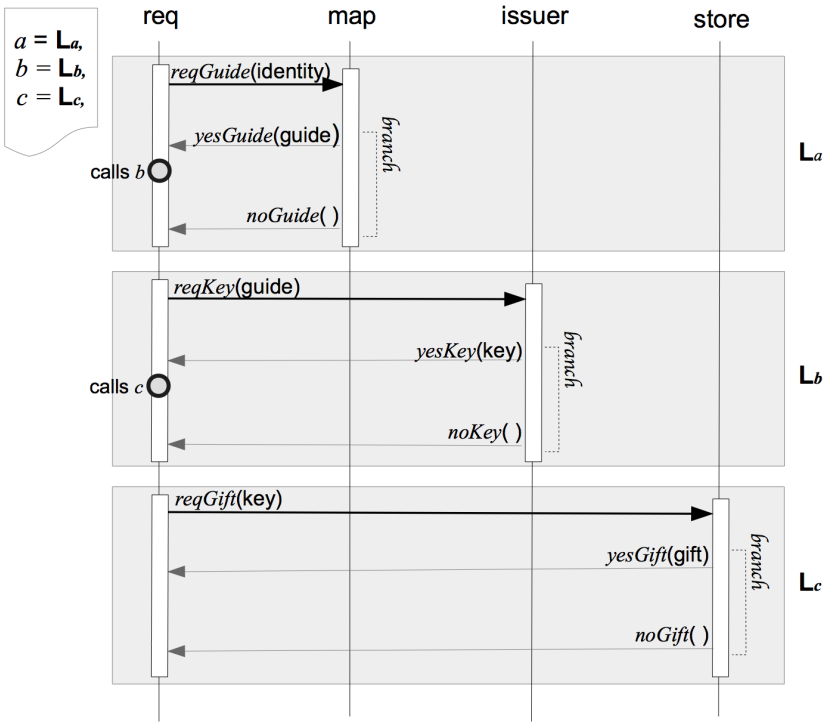

Figure 2: Separated Communications for `req`, `map`, `issuer`, and `store`

unnecessary. If `issuer` is only invoked as the requester gets a *guide* (of type guide) from `map`, and `store` is only invoked as the requester gets the *key* (of type key) from `issuer`, as shown in Figure 2, the interactions become simpler and more readable.

1.2. Approach

With this motivation, [1] proposes *light global types* and a method for *lightening global types* through deleting redundant interactions and, at the same time, generating a set of light global types to keep the communications which are parts of the original protocol (described by a global type). In this way a communication protocol and its participants are invoked only when needed.

We use calls to switch from one communication protocol to another and to keep the same traces of processes when the branch *yesGuide, yesKey*, or *yesGift* is selected. As Figure 2 shows, we let a communication protocol $\mathbf{L}_a$ specify the interactions for getting *guide*, and another protocol $\mathbf{L}_b$ specify the interactions for getting *key* with *guide*. Both *guide* and *key* are knowledge gained from these interactions. *guide* (i.e. a value of type guide) bridges $\mathbf{L}_a$ and $\mathbf{L}_b$ as it is gained in $\mathbf{L}_a$ and used in $\mathbf{L}_b$. The requester then uses *key*, if she successfully got it

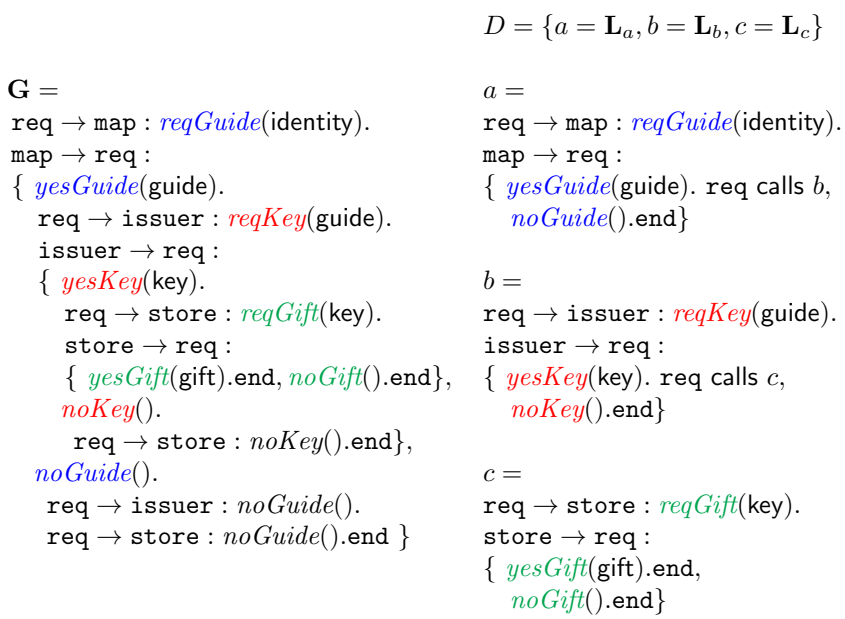

Figure 3: Viewing Interactions As a Whole or As Separated but Related Ones.

in $\mathbf{L}_b$, to gain the wanted gift. Let the communication protocol $\mathbf{L}_c$ implement these final interactions.

As standard we use global types [3, 4] to describe interaction protocols, adding a calls command and declarations for relating protocols. We dub *light global types* the global types written in the extended syntax. In Figure 3 the left-hand-side (LHS) gives the global type representing the protocol in Figure 1 and the right-hand-side (RHS) gives the global types representing the protocols in Figure 2. The difference is in viewing all interactions as one scenario or viewing the interactions as three separated scenarios, connected by commands calls. As usual `store` → `req` : {*yesGift*(gift).end, *noGift*().end} models a communication where `store` sends `req` either the label *yesGift* and a value of type gift or the label *noGift*; in both cases end finishes the interaction. The construct `req` calls b indicates that participant `req` will take care of continuing the interaction following the light global type associated to name b. Note that, although the processes following the LHS and RHS global types in Figure 3 have different behaviours as shown in Figure 1 (where `map`, `issuer` and `store` are all invoked at the same time) and Figure 2 (where `map`, `issuer` and `store` are invoked at different calls), all processes will do essentially the same job no matter which branches have been selected. In other words, as the requester gives the same input to the LHS and RHS maps, the following communications of the LHS and RHS `req`, `map`, `issuer` and `store` are the same, but for the redundant ones.

```
protocol getAll
          (role req, role map,
           role issuer, role store){
  reqGuide(identity) from req to map;
  choice at map{
    yesGuide(guide) from map to req;
    reqKey(guide) from req to issuer;
    choice at issuer{
      yesKey(key) from issuer to req;
      reqGift(key) from req to store;
      choice at store{
        yesGift(gift) from store to req;
      } or {
        noGift() from store to req;
      }
    } or {
      noKey() from issuer to req;
      noKey() from req to store;
    }
  } or {
    noGuide() from map to req;
    noGuide() from req to issuer;
    noGuide() from req to store;
  }
}
```

```
protocol getGuide (role req, role map){
  reqGuide(identity) from req to map;
  choice at map{
    yesGuide(guide) from map to req;
    run protocol
    getKey(role req, role issuer) at req;
  } or {
    noGuide() from map to req;
  }
}

protocol getKey (role req, role issuer){
  reqKey(guide) from req to issuer;
  choice at issuer{
    yesKey(key) from issuer to req;
    run protocol
    getGift(role req, role store) at req;
  } or {
    noKey() from issuer to req;
  }
}

protocol getGift (role req, role store){
  reqGift(key) from req to store;
  choice at store{
    yesGift(gift) from store to req;
  } or {
    noGift() from store to req;
  }
}
```

Figure 4: Scribble for **protocol**s *getAll* (LHS) and *getGuide*, *getKey*, and *getGift* (RHS).

The set of light global types associated to the names a, b and c describes the protocols given by the global type $\mathbf{G}$.

Lightening eases the system costs by removing the interactions which only inform endpoints to terminate or which will not affect their following interactions. We call *redundant* such communications. In the example the communications $\texttt{req} \to \texttt{store} : noKey()$ and $\texttt{req} \to \texttt{issuer} : noGuide().\texttt{req} \to \texttt{store} : noGuide()$ are redundent. Moreover, lightening prevents both local participants and the global network from wasting resources, e.g. keeping online or waiting for step-by-step permissions.

Features of lightening become clearer by looking at the Scribble code implementing the global types in Figure 3, see Figure 4. The keywords are bold. In Scribble, after the keyword **protocol** one writes the protocol's name and declares the roles involved, then the body implements the interactions. The LHS of Figure 4 shows the Scribble code corresponding to the global type $\mathbf{G}$. Although the code is for a simple task, it is complicated since there are three nested **choice**s. On the contrary, the RHS of Figure 4 clearly illustrates the steps for getting a gift in three small protocols (corresponding to the light global types associated to a, b, c, respectively), which are separated but linked (by **run**

and **at**) for preserving the causality. Note that Scribble allows not to declare `issuer` in *getGuide* (*role* `req`, *role* `map`). As explained before, `issuer` is only invoked by **run protocol** *getKey*(*role* `req`, *role* `issuer`) **at** `req`. Similarly, we do not declare `store` in **protocol** *getKey* (*role* `req`, *role* `issuer`).

1.3. Contributions

This work extends [1] by improving and simplifying the lightening functions and by adding:

1. Local types (Definition 3) as projections of light global types and rules of projection (Definition 5).
2. A calculus of asynchronous processes implementing light global types (Figure 5 defines the syntax of processes, Figure 6 defines structural congruence for processes and queues, and Figure 7 defines the reduction rules of processes).
3. A type system (Figure 8 and Figure 9 define typing rules for expressions and processes respectively) for assigning the local types of point 1 to the processes of point 2.
4. We state and prove subject reduction property (Theorem 2) for the type system of point 3.
5. We state and prove (Theorem 3) that a *lightening* process (i.e. a process realising a lightening global type) produces fewer messages (which implies that it has fewer reductions) than the original one; and those fewer messages result exactly from deleting the empty-content messages generated by redundent actions.

Besides *soundness* of lightening (Theorem 1 in [1]) that gives a relation between the sequence of actions (observed by messages) specified in a global type and that specified in its lightening type, Theorem 3 gives a relation between a process welltyped by a global type and a process welltyped by its lightening type. Theorem 3 also shows that a lightening global type can shape more efficient communications among processes by reducing the production of redundent messages.

Paper structure. Section 2 introduces light global types, and Section 3 proposes functions for decomposing global types into light global types. Local types as projections of light global types to endpoints roles are given in Section 2.1. Processes implementing communications described by light global types are presented in Section 4. Section 5 discusses the type system which enables calls of light global types. Section 6 is devoted to the proofs. Section 6.1 shows the soundness (Theorem 1) of the lightening functions, while Section 6.2 shows subject reduction (Theorem 2) of the type system and proves that communications among lightening processes are more efficient (Theorem 3), in the sense that only non-redundent messages are produced, than the non-lightening ones. Finally Section 7 discusses related works.

2. Light Global Types

Light global types are useful to simplify global types by deleting redundant interactions (see Definition 9) which may make a session inefficient. We firstly recall the definition of global types [3], which represent behaviour patterns of communication protocols.

Let $\mathtt{r}, \mathtt{r}_1, \mathtt{r}_2, ...$ range over endpoint roles, $l, l_1, l_2, ...$ range over labels, and $a, b, c, ...$ range over names:

Definition 1 (Standard global types)**.** We define value types, ranged over $S, S', S_i ...$, and global types, ranged over $\mathbf{G}, \mathbf{G}', \mathbf{G}_i, ...$, by the following grammar:

$$\begin{array}{rcl} S & ::= & \mathsf{Str} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Unit} \mid \cdots \\ \mathbf{G} & ::= & \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).\mathbf{G}_j\}_{j \in J} \mid \mathtt{end} \end{array}$$

$S, S', S_i, ...$ include basic types Str, $\mathsf{Bool}, \mathsf{Int}$, and Unit etc. with obvious meanings. The branching type $\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).\mathbf{G}_j\}_{j \in J}$ says that role $\mathtt{r}_1$ sends the label l_j and a message of type S_j to $\mathtt{r}_2$ by selecting $j \in J$ and then the interaction continues as described in $\mathbf{G}_j$. Type $\mathtt{end}$ means termination of the protocol. We write $l()$ as short for $l(\mathsf{Unit})$ and we omit brackets when there is only one branch.

We do not consider recursive global types since redundant interactions cannot be deleted when they occur inside recursion, as discussed in Example 4.

Light global types are global types simply extended with declarations and the construct calls:

Definition 2 (Light global types and declarations)**.** We define light global types, ranged over $\mathbf{L}, \mathbf{L}', \mathbf{L}_i, ...$, and declarations, ranged over $D, D', D_i, ...$, by the following grammar:

$$\begin{array}{rcl} \mathbf{L} & ::= & \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).\mathbf{L}_j\}_{j \in J} \mid \mathbf{G} \mid \mathtt{r}\ \mathsf{calls}\ a \\ D & ::= & \emptyset \mid D, a = \mathbf{L} \end{array}$$

Declarations, denoted by D, are sets which associate names to light global types. For example, in Figure 3 the name a is associated with the type

$$\begin{array}{l} \mathtt{req} \to \mathtt{map} : \mathit{reqGuide}(\mathsf{identity}). \\ \quad \mathtt{map} \to \mathtt{req} : \{\ \mathit{yesGuide}(\mathsf{guide}).\ \mathtt{req}\ \mathsf{calls}\ b, \mathit{noGuide}().\mathtt{end}\ \} \end{array}$$

The type $\mathtt{r}\ \mathsf{calls}\ a$ requires that the set of current declarations contains $a = \mathbf{L}$ for some $\mathbf{L}$. Then role $\mathtt{r}$ continues the session by following the behaviour described in $\mathbf{L}$ and she invites the roles in $\mathbf{L}$. For example, according to the declarations in Figure 3 $\mathtt{req}\ \mathsf{calls}\ b$ prescribes $\mathtt{req}$ to invite $\mathtt{issuer}$ and to continue the session with light global type:

$$\begin{array}{l} \mathtt{req} \to \mathtt{issuer} : \mathit{reqKey}(\mathsf{guide}). \\ \quad \mathtt{issuer} \to \mathtt{req} : \{\ \mathit{yesGuide}(\mathsf{key}).\ \mathtt{req}\ \mathsf{calls}\ c, \mathit{noGuide}().\mathtt{end}\ \} \end{array}$$

2.1. Local Types and Projections

The syntax of local types is standard [5], but for the types $\mathsf{inv}\langle a\rangle$ and $\mathsf{acc}\langle a\rangle$, which are the counterparts of the calls in light global types:

Definition 3 (Local types)**.** We define local types, ranged over T, T', T_i, ..., by the following grammar:

$$T \;::=\; \mathsf{inv}\langle a\rangle \;\mid\; \mathsf{acc}\langle a\rangle \;\mid\; \mathtt{r}!\{l_j(S_j).T_j\}_{j\in J} \;\mid\; \mathtt{r}?\{l_j(S_j).T_j\}_{j\in J} \;\mid\; \mathtt{end}$$

The types $\mathsf{inv}\langle a\rangle$ and $\mathsf{acc}\langle a\rangle$ need a set of declarations D with $D(a)$ defined. The type $\mathsf{inv}\langle a\rangle$ prescribes to invite all roles appearing in $D(a)$ but itself, while the type $\mathsf{acc}\langle a\rangle$ prescribes to accept this invitation.

The sending type $\mathtt{r}!\{l_j(S_j).T_j\}_{j\in J}$ specifies that an endpoint needs to send a message with type selected from $\{l_j(S_j)\}_{j\in J}$ to $\mathtt{r}$.

The receiving type $\mathtt{r}?\{l_j(S_j).T_j\}_{j\in J}$ is dual to the sending type. It specifies that an endpoint will receive a message with type selected from $\{l_j(S_j)\}_{j\in J}$ from $\mathtt{r}$.

Type $\mathtt{end}$ terminates an endpoint's actions.

As usual [3], global and local types are related by projections. Following [5, 6, 7], our definition of projection relies on a merge operator on local types to gather different branches from the same sender.

Definition 4 (Merge of local types)**.**

1. The union $\cup$ of two local types is the partial operator defined by:
   (a) $T \cup T = T$
   (b) $\mathtt{r}?\{l_k(S_k).T_k\}_{k\in I} \cup \mathtt{r}?\{l_k(S_k).T_k\}_{k\in J} = \mathtt{r}?\{l_k(S_k).T_k\}_{k\in I\cup J}$ where $I \cap J = \emptyset$.
2. The merge $\sqcup$ of two local types is the partial operator defined by:
   (a) $T \sqcup T = T$
   (b) $\mathtt{r}?\{l_i(S_i).T_i\}_{i\in I} \;\sqcup\; \mathtt{r}?\{l_j(S'_j).T'_j\}_{j\in J} =$
   $\mathtt{r}?\{l_k(S_k).T_k\}_{k\in I\setminus J} \cup \mathtt{r}?\{l_k(S'_k).T'_k\}_{k\in J\setminus I} \cup \mathtt{r}?\{l_k(S_k).T_k \sqcup T'_k\}_{k\in I\cap J}$
   where $\forall k \in I \cap J, S_k = S'_k$ and $l_i = l_j$ only if $i = j$.

By condition 2a the merge is idempotent. Condition 2b combines two local types receiving messages from the same role $\mathtt{r}$. The resulting local type includes the union of the branches having distinguished labels (i.e. in $I \setminus J$ and $J \setminus I$), and integrates the common labels (i.e., in $I \cap J$). When integrating the common labels, condition 2b makes sure that they have the same sorts (i.e., $S_k = S'_k$).

Our definition of projection differs from that in [5] only for the light global type $\mathtt{r}\ \mathsf{calls}\ a$, which gives $\mathsf{inv}\langle a\rangle$ when projected on $\mathtt{r}$ and $\mathsf{acc}\langle a\rangle$ otherwise.

Definition 5 (Projection of light global types)**.** We define $\mathtt{pj}$ a mapping from a pair of a (light) global type and a role to a local type. $\mathtt{pj}$ projects a (light)

global type to a role and thus generates a corresponding local type:

$$\mathtt{pj}(\mathtt{r}_1 \to \mathtt{r}_2 : \{l_i(S_i).\mathbf{L}_i\}_{i\in I}, \mathtt{r}) = \begin{cases} \mathtt{r}_2!\{l_i(S_i).\mathtt{pj}(\mathbf{L}_i, \mathtt{r})\}_{i\in I} & \textit{if } \mathtt{r} = \mathtt{r}_1, \\ \mathtt{r}_1?\{l_i(S_i).\mathtt{pj}(\mathbf{L}_i, \mathtt{r})\}_{i\in I} & \textit{if } \mathtt{r} = \mathtt{r}_2, \\ \sqcup_{i\in I}\mathtt{pj}(\mathbf{L}_i, \mathtt{r}) & \textit{if } \mathtt{r} \notin \{\mathtt{r}_1, \mathtt{r}_2\} \end{cases}$$

$$\mathtt{pj}(\ \mathtt{r}' \ \mathsf{calls}\ a, \mathtt{r}) = \begin{cases} \mathsf{inv}\langle a\rangle & \textit{if } \mathtt{r} = \mathtt{r}' \\ \mathsf{acc}\langle a\rangle & \textit{otherwise} \end{cases}$$

$$\mathtt{pj}(\mathtt{end}, \mathtt{r}) = \mathtt{end}$$

To project a standard global type to its roles, we only need to apply the first and the third rules in Definition 5.

We end this section by recalling the definition of well-formed standard global type and extending it to declarations.

Definition 6 (Well-formedness)**.**

1. A standard global type is well-formed if it is projectable.
2. A declaration is well formed if all called names do have corresponding entries and all light global types are projectable.

Example 1. Assume $\mathtt{r}_3 \neq \mathtt{r}_1$ and $\mathtt{r}_3 \neq \mathtt{r}_2$. The following global type is well-formed:

$$\mathtt{r}_1 \to \mathtt{r}_2 : \{l_1(S_1).\mathtt{r}_2 \to \mathtt{r}_3 : l_3(S_3).\mathtt{end}, l_2(S_2).\mathtt{r}_2 \to \mathtt{r}_3 : l_4(S_4).\mathtt{end}\}$$

because every participants know how to proceed at each step. Although the label selected by $\mathtt{r}_1 \to \mathtt{r}_2$ is not visible to $\mathtt{r}_3$, $\mathtt{r}_3$ will be informed by $\mathtt{r}_2$ who knows exactly which label has been chosen. In other words, $\mathtt{r}_3$ only needs to wait for a message labelled either with l_3 or l_4 from $\mathtt{r}_2$, and $\mathtt{r}_2$ will send her the right message according to whether l_1 (i.e. then $\mathtt{r}_2$ sends $\mathtt{r}_3$ a message labelled with l_3) or l_2 (i.e. then $\mathtt{r}_2$ sends $\mathtt{r}_3$ a message labelled with l_4) has been selected.

However the following global type is not well-formed:

$$\mathtt{r}_1 \to \mathtt{r}_2 : \{l_1(S_1).\mathtt{r}_3 \to \mathtt{r}_2 : l_3(S_3).\mathtt{end}, l_2(S_2).\mathtt{r}_3 \to \mathtt{r}_2 : l_4(S_4).\mathtt{end}\}$$

because, in this case, $\mathtt{r}_3$ is the sender who does not know either l_1 or l_2 has been selected. $\mathtt{r}_3$ can only send either l_3 or l_4 blindly to $\mathtt{r}_2$, who knows the right branch to proceed. For example, if l_1 has been chosen, but $\mathtt{r}_3$ sends $\mathtt{r}_2$ a message labelled with l_4, then there is a conflict at the receiving of $\mathtt{r}_2$.

3. Lightening Functions

This section describes two functions for removing redundant interactions from (possibly light) global types. It uses lightening, since it adds calls constructors and declarations. In Subsection 6.1 we will show that the protocol obtained by lightening a protocol describes the same non-redundant interactions.

We consider an interaction redundant when only one label can be sent and the message is not meaningful, i.e. it has type Unit and it will not affect the behaviours of the session participants after this interaction. More precisely, we define global contexts:

Definition 7. We define $\mathtt{C}$ as a global context by the following grammar:

$$\mathtt{C} \quad ::= \quad [\,] \mid \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).\mathbf{G}_j, l(S).\mathtt{C}\}_{j \in J}$$

and the mapping *sender* from light global types to sets of roles by:

Definition 8. We define *sender* a mapping from a (light) global type to a set of roles who are *senders* in interactions:

$$\mathit{sender}(\mathbf{L}) \quad = \quad \begin{cases} \bigcup_{i \in I} \mathit{sender}(\mathbf{L}_i) \cup \{\mathtt{r}_1\} & \text{if } \mathbf{L} = \mathtt{r}_1 \to \mathtt{r}_2 : \{l_i(S_i).\mathbf{L}_i\}_{i \in I} \\ \emptyset & \text{otherwise.} \end{cases}$$

Now we formalise the notion of redundant interaction:

Definition 9 (Redundant interaction)**.** The interaction $\mathtt{r}_1 \to \mathtt{r}_2 : l()$ is *redundant* in

$$\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : \; l(\,).\mathbf{L}\,]$$

if $\mathtt{r}_2 \notin \mathit{sender}(\mathbf{L})$.

We require the receiver, $\mathtt{r}_2$, not to be involved in the continuing interactions as a sender. In fact if $\mathtt{r}_2$ as a sender communicates with other participants in $\mathbf{L}$, the message $l(\,)$ can modify the behaviour of $\mathtt{r}_2$.

Example 2. Let

$$\begin{aligned} \mathbf{G}_1 = \mathtt{r}_1 \to \mathtt{r}_2 \quad : \quad \{ & \mathit{req1}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{yes}(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{yes}(\mathsf{Int}).\mathtt{end} \\ & \mathit{req2}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{wait}(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{wait}(\mathsf{Int}).\mathtt{end} \\ & \mathit{req3}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{no}(\,).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{no}(\,).\mathtt{end}\} \end{aligned}$$

then $\mathbf{G}_1$ can be lightened. We can first erase the redundant communication $\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{no}(\,)$. Then the communication $\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{no}(\,)$ becomes redundant, since $\mathtt{r}_3$ is no more involved in the continuing interaction.

Example 3. Let

$$\begin{aligned} \mathbf{G}_2 = \mathtt{r}_1 \to \mathtt{r}_2 \quad : \quad \{ & \mathit{req1}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{yes}(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{yes}(\mathsf{Int}).\mathtt{end} \\ & \mathit{req2}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{wait}(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{wait}(\mathsf{Int}).\mathtt{end} \\ & \mathit{req3}(\mathsf{Int}).\mathtt{r}_1 \to \mathtt{r}_3 : \mathit{no}(\,).\mathtt{r}_3 \to \mathtt{r}_4 : \mathit{no}(\mathsf{Str}).\mathtt{end}\} \end{aligned}$$

then $\mathbf{G}_2$ cannot be lightened. In fact $\mathtt{r}_1 \to \mathtt{r}_3 : no(\ )$ is not redundant as $\mathtt{r}_3$ sends a message to $\mathtt{r}_4$ in the continuing interaction.

Example 4. An example showing that interactions inside recursion cannot be erased is the global type

$$\mu \mathsf{t}.\mathtt{r}_1 \to \mathtt{r}_2 : \{goon(\mathsf{Int}).\mathtt{r}_2 \to \mathtt{r}_3 : goon(\mathsf{Int}).\mathsf{t}, stop().\mathtt{r}_2 \to \mathtt{r}_3 : stop().\mathtt{end}\}$$

The interaction $\mathtt{r}_2 \to \mathtt{r}_3 : stop()$ cannot be erased, since this interaction terminates the recursion, i.e. $\mathtt{r}_2$ tells $\mathtt{r}_3$ that the sequence of integers is ended.

The function $\mathbb{L}$ applied to $\mathbf{L}$ removes the redundant interactions in $\mathbf{L}$ by decomposing $\mathbf{L}$ into separated light global types. The basic idea is that, if one branch contains a redundant interaction whose receiver is not involved in the choice, then all branches are replaced by calls to new declared light global types. Therefore the result of $\mathbb{L}(\mathbf{L})$ is a new light global type and a set of declarations with fresh names.

The function $\mathbb{L}$ is given by induction on the context in which the redundant interaction appears:

Definition 10 (The function $\mathbb{L}$)**.** Define $\mathbb{L}$ a mapping from a (light) global type to a pair of a light global type and its corresponding D, denoted by $(\mathbf{L}, D)$. The application of the function $\mathbb{L}$ to

$$\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L}\ ]$$

where $\mathtt{r}_2 \notin sender(\mathbf{L})$ for eliminating the redundant interaction $\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L}$ is defined by induction on $\mathtt{C}$:

$$\mathbb{L}(\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L}) = (\mathbf{L}, \emptyset)$$

$$\mathbb{L}(\mathtt{r}_1' \to \mathtt{r}_2' : \{l_j(S_j).\mathbf{L}_j, l'(S).\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L}\ ]\}_{j\in J}) =$$

$$\begin{cases} (\mathtt{r}_1' \to \mathtt{r}_2' : \{l_j(S_j).\mathbf{L}_j, l'(S).\mathtt{C}[\mathbf{L}]\}_{j\in J}, \emptyset) & \text{if } \mathtt{r}_2 \in \{\mathtt{r}_1', \mathtt{r}_2'\} \\ (\mathtt{r}_1' \to \mathtt{r}_2' : \{l_j(S_j).\ \mathtt{r}_2'\ \mathsf{calls}\ a_j,\ \ l'(S).\ \mathtt{r}_2'\ \mathsf{calls}\ a\}_{j\in J},\ D') & \text{if } \mathtt{r}_2 \notin \{\mathtt{r}_1', \mathtt{r}_2'\} \\ \qquad \text{where } \mathbb{L}(\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L}\ ]) = (\mathbf{L}', D) & \\ \qquad\quad \text{and } D' = \bigcup_{j\in J}\{a_j = \mathbf{L}_j, a = \mathbf{L}'\} \cup D & \\ \qquad\quad \text{and } a_j, a \text{ are fresh for all } j \in J. & \end{cases}$$

When there is no other branch, i.e. when the context is empty, we simply delete the redundant interaction. In the branching case, if the receiver $\mathtt{r}_2$ of the redundant interaction is also the sender or the receiver of the top branching, then she is aware of the choice of the label l' and the redundant interaction can simply be erased. Otherwise $\mathtt{r}_2$ must receive a communication in all branches l_j for $j \in J$; in this case these communications need to be replaced by calls to fresh names of light global types. We recursively call the mapping $\mathbb{L}$ on

$\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : \ l().\mathbf{L}\ ]$ and take off the redundant interaction $\mathtt{r}_1 \to \mathtt{r}_2 : \ l()$ at the end. Note that the choice between $\mathtt{r}_1'$ and $\mathtt{r}_2'$ for calling (i.e. it does not matter to use $\mathtt{r}_1'$ calls a or $\mathtt{r}_2'$ calls a) is arbitrary, since, if the aimed (light) global type is well-formed, $\mathbb{L}$ does not affect projectability and hence well-formedness of the obtained lightened type.

For instance the application of $\mathbb{L}$ to $\mathbf{G}_1$ as defined in Example 2 to erase $\mathtt{r}_3 \to \mathtt{r}_4 : no(\ )$ gives:

$$(\mathbf{L}, \{a_1 = \mathbf{L}_1, a_2 = \mathbf{L}_2, a_3 = \mathbf{L}_3\}$$

where

$$\begin{array}{lcl}
\mathbf{L} & = & \mathtt{r}_1 \to \mathtt{r}_2 : \{req1(\mathsf{Int}).\ \mathtt{r}_2\ \mathsf{calls}\ a_1, req2(\mathsf{Int}).\ \mathtt{r}_2\ \mathsf{calls}\ a_2, req3(\mathsf{Int}).\ \mathtt{r}_2\ \mathsf{calls}\ a_3\} \\
\mathbf{L}_1 & = & \mathtt{r}_1 \to \mathtt{r}_3 : yes(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : yes(\mathsf{Int}).\mathtt{end} \\
\mathbf{L}_2 & = & \mathtt{r}_1 \to \mathtt{r}_3 : wait(\mathsf{Str}).\mathtt{r}_3 \to \mathtt{r}_4 : wait(\mathsf{Int}).\mathtt{end} \\
\mathbf{L}_3 & = & \mathtt{r}_1 \to \mathtt{r}_3 : no(\ ).\mathtt{end}
\end{array}$$

Note that, a global type can be lightened through $\mathbb{L}$ no matter it is well-formed or not; this is the reason why we set a condition that, if the receiver $\mathtt{r}_2$ of the redundant interaction is not the sender nor the receiver of the top branching, she must receive a communication in all branches l_j for $j \in J$ because the aimed global type may not have balanced branches (see Example 1) which are ensured when the global type is well-formed.

Since the global types in declarations can contain redundant interactions, we can light them by applying recursively the function $\mathbb{L}$. This is formalised by the following function $\mathcal{L}$, whose argument is a set of declarations.

Definition 11 (The function $\mathcal{L}$)**.** Define $\mathcal{L}$ a mapping from D to D:

$$\begin{array}{rcl}
\mathcal{L}(\emptyset) & = & \emptyset \\
\mathcal{L}(\{a = \mathbf{L}\} \cup D) & = & \mathcal{L}(\{a = \mathbf{L}'\} \cup D \cup D') \text{ where } \mathbb{L}(\mathbf{L}) = (\mathbf{L}', D').
\end{array}$$

The function $\mathcal{L}$ is applied recursively to a set of declarations obtaining a new set of declarations until some of the light global types contain redundant interactions. It simply uses the function $\mathbb{L}$ to erase redundant interactions. Notice that we need both $\mathcal{L}$ and $\mathbb{L}$ since they have different domains.

Finally, if the set of declarations contains declarations of the shape $a = \mathtt{end}$, we can simplify it by replacing all $\mathtt{r}$ calls a with $\mathtt{end}$. This can be formalised by the following mapping $\mathcal{D}$ between declarations:

Definition 12. Define $\mathcal{D}$ a mapping from D to D:

$$\mathcal{D}(D \cup \{a = \mathtt{end}\}) = D\{\mathtt{end}/\ \mathtt{r}\ \mathsf{calls}\ a\}$$

assuming that $\mathcal{D}$ leaves D unchanged when D does not contain declarations of the shape $a = \mathtt{end}$.

We consider the global type $\mathbf{G}$ of Figure 3. The application of $\mathbb{L}$ for eliminating the interaction $\mathtt{req} \to \mathtt{store} : noKey()$ gives

$$(\mathbf{L}_a, \{b = \mathbf{L}_b, b' = \mathbf{L}_{b'}, c = \mathbf{L}_c\}),$$

where:

$$\begin{array}{rcl}
\mathbf{L}_a & = & \mathtt{req} \to \mathtt{map} : \mathit{reqGuide}(\mathsf{identity}).\mathtt{map} \to \mathtt{req} : \\
& & \{\ \mathit{yesGuide}(\mathsf{guide}).\ \mathtt{req}\ \mathsf{calls}\ b,\quad \mathit{noGuide}().\ \mathtt{req}\ \mathsf{calls}\ c\} \\
\\
\mathbf{L}_b & = & \mathtt{req} \to \mathtt{issuer} : \mathit{reqKey}(\mathsf{guide}).\mathtt{issuer} \to \mathtt{req} : \\
& & \quad \{\ \mathit{yesKey}(\mathsf{key}).\ \mathtt{req}\ \mathsf{calls}\ b',\ \ \mathit{noKey}().\mathtt{end}\} \\
\\
\mathbf{L}_{b'} & = & \mathtt{req} \to \mathtt{issuer} : \mathit{noGuide}().\mathtt{req} \to \mathtt{store} : \mathit{noGuide}().\mathtt{end} \\
\\
\mathbf{L}_c & = & \mathtt{req} \to \mathtt{store} : \mathit{reqGift}(\mathsf{key}).\mathtt{store} \to \mathtt{req} : \\
& & \{\ \mathit{yesGift}(\mathsf{gift}).\mathtt{end}, \mathit{noGift}().\mathtt{end}\}
\end{array}$$

By applying $\mathcal{L}$ to $\{b = \mathbf{L}_b, b' = \mathbf{L}_{b'}, c = \mathbf{L}_c\}$ to erase the redundant interactions in $\mathbf{L}_{b'}$, we get

$$\{b = \mathbf{L}_b, b' = \mathtt{end}, c = \mathbf{L}_c\}$$

The same result is obtained by using $\mathbb{L}$ to firstly eliminate the interaction

$$\mathtt{req} \to \mathtt{store} : \mathit{noGuide}()$$

At the end, by using the mapping $\mathcal{D}$ to replace $\mathtt{req}\ \mathsf{calls}\ b'$ with $\mathtt{end}$, we get all declarations shown in Figure 3.

It is easy to verify that both $\mathcal{L}$ and $\mathcal{D}$ preserve the well-formedness of sets of declarations, as formalised in the following proposition.

Proposition 1.

1. If $\mathbf{L}$ is well-formed and $\mathbb{L}(\mathbf{L}) = (\mathbf{L}', D)$, then $\mathbf{L}'$ and D is well-formed.
2. If D is well-formed, then both $\mathcal{L}(D)$ and $\mathcal{D}(D)$ are well-formed.

4. Process Calculus

We use $e, e', ...$ to denote expressions, $v, v', ...$ to denote values, $u, u', ...$ to denote both session names $s, s, ...$ and session variables $y, y', ...$

Following [5], Figure 5 defines our calculus for *asynchronous* processes. Processes are ranged over by $P, Q, ...$ and communicate by using two types of channels, i.e. shared channels and session channels. Shared channels, which advertise communication services, are used by processes for initiating a session or calling roles to continue/join a session. Session channels are used for interactions through actions of sending and receiving within established sessions. We use $a, b, c, \ldots$ (already used as names associated to light global types) to denote shared channels, since services can be defined in declarations and they can call each other. Session channels are denoted by $u[\mathtt{r}]$, where u can be either a session name s or a channel variable y and $\mathtt{r}, \mathtt{r}_i$, as before, stand for roles.

Processes $\overline{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P$ and $\widehat{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P$ are for inviting roles to join or continue participating in a session s. Process $\overline{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P$ starts a *new* session, named s, by inviting roles $\mathtt{r}_i$ for $1 \leq i \leq n$ through the

$$
\begin{array}{rcl}
P, Q :: & = & \overline{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P \mid \widehat{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P \mid a(y[\mathtt{r}]).P \mid a\langle s[\mathtt{r}]\rangle.P \\
& & \mid\ u[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e\rangle\rangle.P \mid u[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle \\
& & \mid \mathit{if}\ e\ \mathit{then}\ P\ \mathit{else}\ Q \mid\ P|Q\ \mid s : h \mid (\nu s)P \\
\\
e :: & = & v \mid\ e + e\ \mid\ -e\ \mid\ e \vee e\ \mid\ \neg e ... \\
v :: & = & \mathsf{true}\ \mid\ \mathsf{false}\ \mid\ \mathsf{unit}\ \mid\ 0\ \mid\ 1\ \mid\ ... \\
u :: & = & s \mid\ y \qquad h ::= \varepsilon \mid\ h \cdot m \qquad m ::= \langle \mathtt{r}_i, \mathtt{r}_j, l\langle v\rangle\rangle \\
E :: & = & [\,] \mid\ (\nu s)E\ \mid\ E \mid P
\end{array}
$$

Figure 5: Syntax of Processes.

$$
\begin{array}{c}
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu s)P \mid Q \equiv (\nu s)(P \mid Q)\ \mathit{if}\ s \notin \mathsf{fn}(Q) \quad (\nu s s')P \equiv (\nu s' s)P \quad (\nu s)\mathbf{0} \equiv \mathbf{0} \\
\varepsilon \cdot h \equiv h \equiv h \cdot \varepsilon \quad \dfrac{m_1 \cdot m_2 \frown m_2 \cdot m_1}{h \cdot m_1 \cdot m_2 \cdot h' \equiv h \cdot m_2 \cdot m_1 \cdot h'} \quad \dfrac{h \equiv h'}{s : h \equiv s : h'}
\end{array}
$$

Figure 6: Structural Congruence for Processes and Queues.

shared channel a, then it continues as P. Similarly process $\widehat{a}\langle s, \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}\rangle.P$ invites roles $\mathtt{r}_i$ for $1 \leq i \leq n$ to continue or join the already *established* session s.

Process $a(y[\mathtt{r}]).P$ joins the session s by playing role $\mathtt{r}$ through the shared channel a. Similarly process $a\langle s[\mathtt{r}]\rangle.P$ continues to play role $\mathtt{r}$ in the session s.

Processes $u[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e\rangle\rangle.P$ and $u[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle$ communicate using channel $u[\mathtt{r}_1]$. Process $u[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e\rangle\rangle.P$ sends a message with value e tagged with label l to the process playing role $\mathtt{r}_2$ in the same session; symmetrically, process $u[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle$ is ready to receive a message from $\mathtt{r}_2$ tagged with a label l_k for some $k \in I$.

We need message queues [3], since processes communicate in an asynchronous way. $h, h', ...$ stand for run-time queues, which may be empty (notation ε) or contain messages ranged over $m, m', ...$ A message m has a shape $\langle \mathtt{r}_1, \mathtt{r}_2, l\langle v\rangle\rangle$ for delivering a value: this message is sent from $\mathtt{r}_1$ to $\mathtt{r}_2$ with content v tagged with label l. Process $s : h$ tells that h is the queue of session s.

$E, E', ...$ stand for evaluation contexts, which are either holes, restrictions of contexts, or parallel compositions of contexts with processes.

Structural congruence for processes and queues is defined in Figure 6. Following [3] the messages m_1 and m_2 are *permutable* (notation $m_1 \cdot m_2 \frown m_2 \cdot m_1$) if $m_1 = \langle \mathtt{r}_1, \mathtt{r}_2, * \rangle$ and $m_2 = \langle \mathtt{r}'_1, \mathtt{r}'_2, * \rangle$, and ($\mathtt{r}_1 \neq \mathtt{r}'_1$ or $\mathtt{r}_2 \neq \mathtt{r}'_2$) where $*$ means that the content can be any.

Figure 7 defines the reduction rules of processes.

$$\overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P \mid a(y_1[\mathtt{r}_1]).P_1 \mid ... \mid a(y_n[\mathtt{r}_n]).P_n \longrightarrow (\nu s)(P \mid P_1\{s/y_1\} \mid ... \mid P_n\{s/y_n\} \mid \ s : \varepsilon) \qquad \textsf{[link]}$$

$$(\nu s)(\widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_m, \mathtt{r}_{m+1}..., \mathtt{r}_n\}\rangle.P \mid a\langle s[\mathtt{r}_1]\rangle.P_1 \mid \ \ldots \ \mid a\langle s[\mathtt{r}_m]\rangle.P_m \mid s : \varepsilon) \mid a(y_1[\mathtt{r}_{m+1}]).P_{m+1} \mid ... \mid a(y_{n-m}[\mathtt{r}_n]).P_n \longrightarrow (\nu s)(P \mid P_1 \mid ... \mid P_m \mid P_{m+1}\{s/y_1\} \mid ... \mid P_n\{s/y_{n-m}\} \mid s : \varepsilon) \qquad \textsf{[call]}$$

$$s[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e\rangle\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle v\rangle\rangle \quad e \downarrow v \qquad \textsf{[sel]}$$

$$s : \langle \mathtt{r}_2, \mathtt{r}_1, l_k\langle v\rangle\rangle \cdot h \mid s[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle \longrightarrow s : h \mid P_k\{v/x_k\} \quad k \in I \qquad \textsf{[bra]}$$

$$\textit{if}\ \mathsf{true}\ \textit{then}\ P\ \textit{else}\ Q \longrightarrow P \quad \textit{if}\ \mathsf{false}\ \textit{then}\ P\ \textit{else}\ Q \longrightarrow Q \qquad \textsf{[if]}$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} \qquad \frac{P \longrightarrow P'}{E[P] \longrightarrow E[P']} \qquad \textsf{[str/ctx]}$$

Figure 7: Reduction Rules of Processes.

The reduction rule **[link]** is standard [3]. Process $\overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P$ starts a new session s by inviting $a(y_i[\mathtt{r}_i]).P_i$ where $i \leq i \leq n$. The reduction creates a new empty queue $s : \varepsilon$, substitutes y_i with s in P_i for $1 \leq i \leq n$ and makes s private.

The reduction rule **[call]** is peculiar to our calculus. The processes in the first line, i.e. the processes playing the roles $\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_m$, already share the session s. The processes in the second line, i.e. the processes playing the roles $\mathtt{r}_{m+1}, ..., \mathtt{r}_n$, instead join the session after the reduction. In this way more roles are added to an already established session. Notice that the queue must be empty, since a new light global type (associated to the name a) will prescribe the successive communications.

The reduction rule **[sel]** says that process $s[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e\rangle\rangle.P$ puts the message $\langle \mathtt{r}_1, \mathtt{r}_2, l\langle v\rangle\rangle$ (where $e \downarrow v$) in the queue $s : h$, and then P is ready for the next action. Symmetrically, the reduction rule **[bra]** says that process $s[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle$ receives the top message from $s : \langle \mathtt{r}_2, \mathtt{r}_1, l_k\langle v\rangle\rangle \cdot h$ under the condition $k \in I$, and the process at branch l_k proceeds after substituting every x_k with v in P_k.

5. Typing Rules

For defining a type system, we firstly define shared environments, ranged over $\Gamma, \Gamma', \Gamma_i, ...$, by the following grammar:

Definition 13. A shared environment Γ is a (possibly empty) finite mapping from variables to their sorts

$$\Gamma ::= \emptyset \ \mid \ \Gamma, x : S$$

where the notation $\Gamma, x : S$ means that x does not occur in Γ.

$$\Gamma, x : S \vdash x : S \qquad \Gamma \vdash \mathsf{unit} : \mathsf{Unit} \qquad \Gamma \vdash 0, 1, ... : \mathsf{Int} \quad \textsf{[T-axiom/unit/int]}$$

$$\Gamma \vdash \mathsf{true}, \mathsf{false} : \mathsf{Bool} \quad \frac{\Gamma \vdash e_i : \mathsf{Bool} \quad i \in \{1, 2\}}{\Gamma \vdash e_1 \vee e_2 : \mathsf{Bool}} \qquad \textsf{[T-bool/or]}$$

Figure 8: Typing Rules for Expressions.

We assume that expressions are typed by sorts, as usual. The typing judgments for expressions are of the shape

$$\Gamma \vdash e : S$$

and the typing rules for expressions are standard, see Figure 8.

In order to type queues we introduce message types as usual [3]. A message type $\mathtt{M}$ is a (possibly empty) finite sequence of types of the shape $\langle \mathtt{r}_1, \mathtt{r}_2, l\langle S\rangle\rangle$:

Definition 14 (Message types)**.** We define messages types, ranged over $\mathtt{M}, \mathtt{M}', \mathtt{M}_i, ...$, by the following grammar:

$$\mathtt{M} \quad ::= \quad \epsilon \mid \langle \mathtt{r}_1, \mathtt{r}_2, l\langle S\rangle\rangle \mid \mathtt{M} \cdot \mathtt{M}$$

$\langle \mathtt{r}_1, \mathtt{r}_2, l\langle S\rangle\rangle$ types a message which is sent from $\mathtt{r}_1$ to $\mathtt{r}_2$ and carries a value of type S with label l.

Processes are typed by session environments ranged over $\Delta, \Delta', ...$:

Definition 15. Session environments are (possibly empty) finite mappings from session channels to local types and from session queues to message types:

$$\Delta ::= \emptyset \;\mid\; \Delta, u[\mathtt{r}] : T \;\mid\; \Delta, s : \mathtt{M}$$

Definition 16. If $dom(\Delta) \cap dom(\Delta') = \emptyset$, then $\Delta \cup \Delta' = \Delta, \Delta'$.

Definition 17. We say Δ is end only if and only if $\Delta(u[\mathtt{r}]) = \mathtt{end}$ for any $u[\mathtt{r}] \in dom(\Delta)$.

In typing processes we need to take into account declarations. Therefore typing judgments for processes are of the shape:

$$\Gamma \vdash_D P \rhd \Delta$$

If $\mathbf{L}$ is a light global types with D, we type processes implementing this communication protocol using the set of declarations $\{a = \mathbf{L}\} \cup D$, where a is a fresh name.

Similarly to the definition of *sender*, for convenience, we define *role* to generate the set of roles in a (light) global type:

$$\frac{\Delta \text{ end only}}{\Gamma \vdash_D \mathbf{0} \triangleright \Delta} \quad \textsf{[T-idle]}$$

$$\frac{\Gamma \vdash_D P \triangleright \Delta, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r}) \quad \{\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_n\} = \mathit{role}(D(a))}{\Gamma \vdash_D \overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P \triangleright \Delta} \quad \textsf{[T-mcast]}$$

$$\frac{\Gamma \vdash_D P \triangleright \Delta, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r}) \quad \{\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_n\} = \mathit{role}(D(a))}{\Gamma \vdash_D \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P \triangleright \Delta, s[\mathtt{r}] : \mathsf{inv}\langle a\rangle} \quad \textsf{[T-call]}$$

$$\frac{\Gamma \vdash_D P \triangleright \Delta, y[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})}{\Gamma \vdash_D a(y[\mathtt{r}]).P \triangleright \Delta} \quad \textsf{[T-acc-new]}$$

$$\frac{\Gamma \vdash_D P \triangleright \Delta, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})}{\Gamma \vdash_D a\langle s[\mathtt{r}]\rangle.P \triangleright \Delta, s[\mathtt{r}] : \mathsf{acc}\langle a\rangle} \quad \textsf{[T-acc-cont]}$$

$$\frac{\Gamma \vdash e : S_k \quad \Gamma \vdash_D P \triangleright \Delta, u[\mathtt{r}_1] : T_k \quad k \in I}{\Gamma \vdash_D u[\mathtt{r}_1]!\langle \mathtt{r}_2, l_k\langle e\rangle\rangle.P \triangleright \Delta, u[\mathtt{r}_1] : \mathtt{r}_2!\{l_i(S_i).T_i\}_{i\in I}} \quad \textsf{[T-sel]}$$

$$\frac{\forall j \in I : \Gamma, x_j : S_j \vdash_D P_j \triangleright \Delta, u[\mathtt{r}_1] : T_j}{\Gamma \vdash_D u[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i\}_{i\in I}\rangle \triangleright \Delta, u[\mathtt{r}_1] : \mathtt{r}_2?\{l_i(S_i).T_i\}_{i\in I}} \quad \textsf{[T-bra]}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash_D P \triangleright \Delta \quad \Gamma \vdash_D Q \triangleright \Delta}{\Gamma \vdash_D \mathit{if}\ e\ \mathit{then}\ P\ \mathit{else}\ Q \triangleright \Delta} \quad \textsf{[T-if]}$$

$$\frac{\Gamma \vdash_D P_1 \triangleright \Delta_1 \quad \Gamma \vdash_D P_2 \triangleright \Delta_2}{\Gamma \vdash_D P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \qquad \Gamma \vdash_D s : \varepsilon \triangleright \{s : \epsilon\} \quad \textsf{[T-par/-qnull]}$$

$$\frac{\Gamma \vdash v : S \quad \Gamma \vdash_D s : h \triangleright \Delta, s : \mathtt{M}}{\Gamma \vdash_D s : h \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle v\rangle\rangle \triangleright \Delta, s : \mathtt{M} \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle S\rangle\rangle} \quad \textsf{[T-q]}$$

$$\frac{\Gamma \vdash_D P \triangleright \Delta \quad \Delta \approx \Delta'}{\Gamma \vdash_D P \triangleright \Delta'} \qquad \frac{\Gamma \vdash_D P \triangleright \Delta \qquad \Delta \text{ coherent for } s}{\Gamma \vdash_D (\nu s)P \triangleright \Delta \setminus s} \quad \textsf{[T-equiv/new]}$$

Figure 9: Typing Rules for Processes.

Definition 18. We define *role* a mapping from a (light) global type to a set of roles who are involved in interactions:

$$\mathit{role}(\mathbf{L}) \quad = \quad \begin{cases} \bigcup_{i\in I} \mathit{role}(\mathbf{L}_i) \cup \{\mathtt{r}_1, \mathtt{r}_2\} & \mathit{if}\ \mathbf{L} = \mathtt{r}_1 \to \mathtt{r}_2 : \{l_i(S_i).\mathbf{L}_i\}_{i\in I} \\ \emptyset & \mathit{otherwise.} \end{cases}$$

Figure 9 gives the typing rules for processes. An idle process is typed by an end-only session environment (rule [T-idle]).

Rule [T-mcast] types a process which initiates a new session on the shared name a, where a is the name associated by the set of declarations D to the light type which prescribes the communications inside the session. If $\{\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_n\}$

are the roles of $D(a)$, then $\mathtt{r}$ invites the roles $\{\mathtt{r}_1, ..., \mathtt{r}_n\}$ to join the session. Rule [T-call] types a process which invites some roles to continue an already started session and new roles to join the session. The session environment associates the type $\mathsf{inv}\langle a\rangle$ to the session channel of the process. A process accepting to join a session is typed with the empty session environment (rule [T-acc-new]). Instead a process accepting to continue a session on channel $s[\mathtt{r}]$ is typed by the environment which associates the type $\mathsf{acc}\langle a\rangle$ to $s[\mathtt{r}]$ (rule [T-acc-cont]). Note that the premises of these four rules require that $D(a)$ is defined and the session channel can be typed with the projection.

Rules [T-sel] and [T-bra] are standard for typing output and input processes [5].

Rule [T-par] types the parallel composition of two processes. Note that $dom(\Delta) \cap dom(\Delta') = \emptyset$ based on Definition 16.

The typing rules for the conditional and the queues are intuitive.

Similarly to the permutation of messages (see Figure 6) we say that two message types $\mathtt{M}_1$ and $\mathtt{M}_2$ are *permutable*, denoted by $\mathtt{M}_1 \cdot \mathtt{M}_2 \curvearrowright \mathtt{M}_2 \cdot \mathtt{M}_1$, if $\mathtt{M}_1 = \langle \mathtt{r}_1, \mathtt{r}_2, * \rangle$ and $\mathtt{M}_2 = \langle \mathtt{r}'_1, \mathtt{r}'_2, * \rangle$, and ($\mathtt{r}_1 \neq \mathtt{r}'_1$ or $\mathtt{r}_2 \neq \mathtt{r}'_2$), where $*$ means that the type of the content can be any.

We define the equivalence $\equiv$ on message types by:

$$\frac{\mathtt{M}_1 \cdot \mathtt{M}_2 \curvearrowright \mathtt{M}_2 \cdot \mathtt{M}_1}{\mathtt{M} \cdot \mathtt{M}_1 \cdot \mathtt{M}_2 \cdot \mathtt{M}' \equiv \mathtt{M} \cdot \mathtt{M}_2 \cdot \mathtt{M}_1 \cdot \mathtt{M}'}$$

Session environments are considered modulo equivalence of message types and disregarding the type $\mathtt{end}$. This is formalised in the following definition of equivalence $\approx$ between session environments, which is used in rule [T-equiv].

Definition 19 (Equivalence on session environments)**.** We define $\Delta_1 \approx \Delta_2$ by:

1. $u[\mathtt{r}] \in dom(\Delta_1) \cap dom(\Delta_2)$ implies $\Delta_1(u[\mathtt{r}]) = \Delta_2(u[\mathtt{r}])$ and
2. $u[\mathtt{r}] \in dom(\Delta_i)$ and $u[\mathtt{r}] \notin dom(\Delta_j)$ with $i, j \in \{1, 2\}$ and $i \neq j$ imply $\Delta_i(u[\mathtt{r}]) = \mathtt{end}$ and
3. $s \in dom(\Delta_i)$ implies $s \in dom(\Delta_j)$ and $\Delta_1(s) \equiv \Delta_2(s)$ for $i, j \in \{1, 2\}$ and $i \neq j$.

In rule [T-news], $\Delta \setminus s$ is defined as expected:

$$\Delta \setminus s = \{s'[\mathtt{r}] : T \mid s'[\mathtt{r}] : T \in \Delta, s' \neq s\} \cup \{s' : \mathtt{M} \mid s' : \mathtt{M} \in \Delta, s' \neq s\}$$

As usual [3] the restriction of a session name can be typed only if in the session environment the types of the relative channels and of the queue offer dual communications. In this case we say that the session environment is *coherent* for the session name. In order to formalise the property of *coherence*, we need to single out the communication offered by each role to the other ones. This is done by defining:

1. the projection of a message type on a role, which only keeps the messages having that role as receiver (Definition 20);

2. the session remainder, which erases from a local type the inputs corresponding to the messages in the projection of a message type (Definition 21);
3. the projection of a local type on a role, which only keeps the communication actions having that role as sender or as receiver (Definition 24).

Then a session evironment is coherent *for session* s if all session remainders regarding s are defined (condition (1a) of Definition 26) and all projections regarding s offer dual communications (condition (1b) of Definition 26). A session environment is coherent if it is coherent for any sessions appearing in it.

Definition 20 (Projection of message types)**.** The projection of message type $\mathtt{M}$ on a role $\mathtt{r}$ (notation $\mathtt{M}\upharpoonright\mathtt{r}$) is defined by:

$$\epsilon\upharpoonright\mathtt{r} = \epsilon \qquad (\langle \mathtt{r}_1, \mathtt{r}_2, l\langle S\rangle\rangle \cdot \mathtt{M})\upharpoonright\mathtt{r} = \begin{cases} \langle\!\langle \mathtt{r}_1, l\langle S\rangle\rangle\!\rangle \cdot \mathtt{M}\upharpoonright\mathtt{r} & \textit{if } \mathtt{r} = \mathtt{r}_2 \\ \mathtt{M}\upharpoonright\mathtt{r} & \textit{otherwise} \end{cases}$$

Projections of message types (dubbed *projection types*) are ranged over by π and generated by:

$$\pi ::= \epsilon \mid \langle\!\langle \mathtt{r}, l\langle S\rangle\rangle\!\rangle \mid \pi \cdot \pi$$

Following [8, 9], we define the session remainder of a local type T and a projection type π as the local type obtained from T by erasing all branches that have corresponding selections in π. Clearly the session remainder is defined only if T and π agree on labels and on types of exchanged values.

Definition 21 (Session remainder)**.** We define operator $-$ as a mapping from a pair of (T, π) to T. It results a local type after absorbing coming messages at a local type:

$$\textsf{[tr-qnull]}\ \ T - \epsilon = T$$

$$\textsf{[tr-bra]}\ \frac{T_k - \pi = T' \quad k \in I}{\mathtt{r}?\{l_i(S_i).T_i\}_{i\in I} - \langle\!\langle \mathtt{r}, l_k\langle S_k\rangle\rangle\!\rangle \cdot \pi = T'}$$

$$\textsf{[tr-sel]}\ \frac{\forall i \in I:\ \ T_i - \pi = T_i'}{\mathtt{r}!\{l_i(S_i).T_i\}_{i\in I} - \pi = \mathtt{r}!\{l_i(S_i).T_i'\}_{i\in I}}$$

Notice that the remainder of $\mathsf{inv}\langle a\rangle$ and $\mathsf{acc}\langle a\rangle$ are defined only if π is ϵ. This means that the queue must be empty before starting the execution of a new light global type.

In order to represent the communications with *a specific role*, we define:

Definition 22. Define *communication types*, ranged over by $\mathtt{T}$, by the following grammar:

$$\mathtt{T} \ ::= \ !\{l_i(S_i).\mathtt{T}_i\}_{i\in I} \ \mid \ ?\{l_i(S_i).\mathtt{T}_i\}_{i\in I} \ \mid \ \mathtt{end}$$

On communication types we define a merge operator, denoted by $\sqcup$, which is similarly to the merge operator on local types (see Definitions 4).

Definition 23 (Merge of communication types)**.** Let $\dagger \in \{!, ?\}$.

1. The union $\bigcup$ of two communication types is the following partial operator:
   (a) $\mathtt{T} \bigcup \mathtt{T} = \mathtt{T}$.
   (b) $\dagger\{l_k(S_k).\mathtt{T}_k\}_{k\in I} \bigcup \dagger\{l_k(S_k).\mathtt{T}_k\}_{k\in J} = \dagger\{l_k(S_k).T_k\}_{k\in I\cup J}$ where $I \cap J = \emptyset$.
2. The merge $\sqcup$ of two communication types is the following partial operator:
   (a) $\mathtt{T} \sqcup \mathtt{T} = \mathtt{T}$.
   (b) $\dagger\{l_i(S_i).\mathtt{T}_i\}_{i\in I} \sqcup \dagger\{l_j(S'_j).\mathtt{T}'_j\}_{j\in J} =$
   $\dagger\{l_k(S_k).\mathtt{T}_k\}_{k\in I\setminus J} \bigcup \dagger\{l_k(S'_k).\mathtt{T}'_k\}_{k\in J\setminus I} \bigcup \dagger\{l_k(S_k).\mathtt{T}_k \sqcup \mathtt{T}'_k\}_{k\in I\cap J}$
   where $\forall k \in I \cap J, S_k = S'_k$ and $l_i = l_j$ only if $i = j$.

We can now single out, from a local type, the communications toward a fixed role by means of the projection function `pjl`. Notice that `pjl` only takes into account the explicit communications, not those that could be done after an $\mathsf{inv}\langle a\rangle$ or an $\mathsf{acc}\langle a\rangle$ type.

Definition 24 (Projection of local types)**.** Let $\dagger \in \{!, ?\}$. We define `pjl` a mapping from a pair of a local type and a role to a type toward that role.

$$\mathtt{pjl}(\mathtt{r}' \dagger \{l_i(S_i).T_i\}_{i\in I}, \mathtt{r}) = \begin{cases} \dagger\{l_i(S_i).\mathtt{pjl}(T_i, \mathtt{r}')\}_{i\in I} & \text{if } \mathtt{r} = \mathtt{r}' \\ \bigsqcup_{i\in I} \mathtt{pjl}(T_i, \mathtt{r}) & \text{otherwise.} \end{cases}$$

$$\mathtt{pjl}(\mathsf{inv}\langle a\rangle, \mathtt{r}) = \mathtt{pjl}(\mathsf{acc}\langle a\rangle, \mathtt{r}) = \mathtt{pjl}(\mathtt{end}, \mathtt{r}) = \mathtt{end}$$

The duality relation $\bowtie$ on communication types is defined as usual [3]:

Definition 25 (Duality)**.** We define $\bowtie$ as a binary relation between local types:

$$\frac{\mathtt{T}_i \bowtie \mathtt{T}'_i}{!\{l_i(S_i).\mathtt{T}_i\}_{i\in I} \bowtie ?\{l_i(S_i).\mathtt{T}_i{}'\}_{i\in I}} \qquad \mathtt{end} \bowtie \mathtt{end}$$

It is easy to verify that if we first project a light global type and then we project the obtained local types by exchanging two roles, we get dual communication types.

Proposition 2. If $\mathtt{r}_1, \mathtt{r}_2 \in \mathit{role}(\mathbf{L})$ and $\mathtt{r}_1 \neq \mathtt{r}_2$ and $\mathtt{pj}(\mathbf{L}, \mathtt{r}_1)$ and $\mathtt{pj}(\mathbf{L}, \mathtt{r}_2)$ are defined, then $\mathtt{pjl}(\mathtt{pj}(\mathbf{L}, \mathtt{r}_1), \mathtt{r}_2) \bowtie \mathtt{pjl}(\mathtt{pj}(\mathbf{L}, \mathtt{r}_2), \mathtt{r}_1)$ for any $\mathbf{L}$.

This proposition says that the endpoint types obtained from a shared global type are always dual to each other. It is intuitive since, by meaning, a global type describes interactions, which are always composed by a sender and a receiver.

Lastly we are able to define the property of coherence of session environments:

Definition 26. 1. A session environment Δ is coherent for s if:

(a) $s[\mathtt{r}] : T, s : \mathtt{M} \in \Delta$ imply that $T - \mathtt{M}\!\upharpoonright\!\mathtt{r}$ is defined;

(b) $s[\mathtt{r}_i] : T_i, s[\mathtt{r}_j] : T_j, s : \mathtt{M} \in \Delta$ with $i \neq j$ imply that

$$\mathtt{pjl}(T_i - \mathtt{M}\!\upharpoonright\!\mathtt{r}_i, \mathtt{r}_j) \bowtie \mathtt{pjl}(T_j - \mathtt{M}\!\upharpoonright\!\mathtt{r}_j, \mathtt{r}_i).$$

2. A session environment Δ is coherent if it is coherent for all $s \in \mathit{dom}(\Delta)$.

6. Properties

6.1. Safety of the Lightening Functions

In order to discuss the correctness of our lightening function, following [10] we view light global types with relative sets of declarations as denoting languages of communications which can occur in multiparty sessions. More formally the following definition gives a labelled transition system for light global types with respect to a fixed set of declarations. As usual τ means a silent move, and $\checkmark$ means session termination.

Definition 27 (LTS)**.**

$$\frac{}{\mathtt{r}\ \mathsf{calls}\ a \xrightarrow{\tau}_D \mathbf{L}}\ [call] \qquad \frac{\mathtt{r}_2 \notin \mathit{sender}(\mathbf{L})}{\mathtt{r}_1 \to \mathtt{r}_2 :\ l().\mathbf{L} \xrightarrow{\tau}_D \mathbf{L}}\ [red] \qquad \frac{}{\mathtt{end} \xrightarrow{\checkmark}_D}\ [end]$$

$$\frac{J \neq \{j\} \text{ or } S_j \neq \mathsf{Unit} \text{ or } \mathtt{r}_2 \in \mathit{sender}(\mathbf{L})}{\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).\mathbf{L}_j\}_{j\in J} \xrightarrow{\mathtt{r}_1\to\mathtt{r}_2:l_j(S_j)}_D \mathbf{L}_j}\ [act]$$

We convene that λ ranges over $\mathtt{r}_1 \to \mathtt{r}_2 : l(S)$ and $\checkmark$, and that σ ranges over sequences of λ. We use the standard notation:

$$\begin{array}{rcl}
\mathbf{L} \overset{\lambda}{\Longrightarrow}_D \mathbf{L}' & \text{if} & \mathbf{L} \xrightarrow{\tau}{}^*_D \mathbf{L}_1 \xrightarrow{\lambda}_D \mathbf{L}_2 \xrightarrow{\tau}{}^*_D \mathbf{L}' \text{ for some } \mathbf{L}_1, \mathbf{L}_2 \\
\mathbf{L} \overset{\lambda\cdot\sigma}{\Longrightarrow}_D \mathbf{L}' & \text{if} & \mathbf{L} \overset{\lambda}{\Longrightarrow}_D \mathbf{L}_1 \overset{\sigma}{\Longrightarrow}_D \mathbf{L}' \text{ for some } \mathbf{L}_1 \\
\mathbf{L} \overset{\sigma\cdot\checkmark}{\Longrightarrow}_D & \text{if} & \mathbf{L} \overset{\sigma}{\Longrightarrow}_D \mathbf{L}_1 \overset{\checkmark}{\Longrightarrow}_D \text{ for some } \mathbf{L}_1
\end{array}$$

The language generated by $\mathbf{L}$ and relative to D is the set of sequences σ obtained by reducing $\mathbf{L}$ using D. We take this language as the meaning of $\mathbf{L}$ relative to D (notation $[\![\mathbf{L}]\!]_D$).

Definition 28 (Semantics)**.** $[\![\mathbf{L}]\!]_D = \{\sigma \mid \mathbf{L} \overset{\sigma}{\Longrightarrow}_D \mathbf{L}' \text{ for some } \mathbf{L}' \text{ or } \mathbf{L} \overset{\sigma}{\Longrightarrow}_D\}$.

Soundness of lightening then amounts to show that the functions $\mathbb{L}$ and $\mathcal{L}$ preserve the meaning of light global types with respect to the relative sets of declarations.

Theorem 1 (Soundness)**.** 1. Let $\mathbf{L}$ be a light global type with set of declarations D. If $\mathbb{L}(\mathbf{L}) = (\mathbf{L}', D')$, then $[\![\mathbf{L}]\!]_D = [\![\mathbf{L}']\!]_{D'\cup D}$.

2. If $\mathcal{L}(D) = D'$ and $a \in dom(D)$, then $[\![D(a)]\!]_D = [\![D'(a)]\!]_{D'}$.

Proof. 1. The proof is by induction on $\mathbf{L}$ and by cases on Definition 10.

(a) If $\mathbf{L} = \mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}''$, then $\mathbb{L}(\mathbf{L}) = (\mathbf{L}'', \emptyset)$. We conclude since by Definition 28 and by rule [*red*] $[\![\mathbf{L}]\!]_D = [\![\mathbf{L}'']\!]_D$.

(b) Let $\mathbf{L} = \mathtt{C}_0[\mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}'']$ where $\mathtt{C}_0 = \mathtt{r}'_1 \to \mathtt{r}'_2 : \{l_j(S_j).\mathbf{L}_j, l'(S).\mathtt{C}[\,]\}_{j\in J}$. By Definition 28 and by rule [*act*]

$$\begin{array}{rcl} [\![\mathbf{L}]\!]_D & = & \bigcup_{j\in J}\{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j)\cdot\sigma \mid \sigma \in [\![\mathbf{L}_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S)\cdot\sigma \mid \sigma \in [\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}'']]\!]_D\ \} \end{array}$$

i. If $\mathtt{r}_2 \in \{\mathtt{r}'_1, \mathtt{r}'_2\}$, then

$$\begin{array}{rcl} [\![\mathbf{L}']\!]_D & = & \bigcup_{j\in J}\{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j)\cdot\sigma \mid \sigma \in [\![\mathbf{L}_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S)\cdot\sigma \mid \sigma \in [\![\mathtt{C}[\mathbf{L}'']]\!]_D\ \} \end{array}$$

We conclude since rule [*red*] $[\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}'']]\!]_D = [\![\mathtt{C}[\mathbf{L}'']]\!]_D$.

ii. If $\mathtt{r}_2 \notin \{\mathtt{r}'_1, \mathtt{r}'_2\}$, then, by rule [*call*], we have

$$\begin{array}{rcl} [\![\mathbf{L}']\!]_{D'\cup D} & = & \bigcup_{j\in J}\{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j)\cdot\sigma \mid \sigma \in [\![\ \mathtt{r}'_2\ \mathsf{calls}\ a_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S)\cdot\sigma \mid \sigma \in [\![\ \mathtt{r}'_2\ \mathsf{calls}\ a]\!]_{D'\cup D}\ \} \\ & = & \bigcup_{j\in J}\{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j)\cdot\sigma \mid \sigma \in [\![\mathbf{L}_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S)\cdot\sigma \mid \sigma \in [\![\mathbf{L}'_0]\!]_{D'\cup D}\ \} \end{array}$$

since we have

$$\begin{array}{rcl} D' & = & \bigcup_{j\in J}\{a_j = \mathbf{L}_j, a = \mathbf{L}'_0\}_{i\in J} \cup D'_0 \\ (\mathbf{L}'_0, D'_0) & = & \mathbb{L}(\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}'']) \end{array}$$

We conclude since by induction

$$[\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().\mathbf{L}'']]\!]_D = [\![\mathbf{L}'_0]\!]_{D'_0\cup D} = [\![\mathbf{L}'_0]\!]_{D'\cup D}.$$

2. Let $D \setminus a = \{b = \mathbf{L} \mid b = \mathbf{L} \in D \ \&\ a \neq b\}$. By definition of $\mathcal{L}$ there are $a_1, \ldots, a_m, \mathbf{L}_1, \ldots, \mathbf{L}_n, D_1, \ldots, D_n, D'_0, \ldots, D'_n$ such that:
   (a) $D'_0 = D$
   (b) $\mathbb{L}(D'_i(a_{j_i})) = (\mathbf{L}_{i+1}, D_{i+1})$
   (c) $D'_{i+1} = D_{i+1} \cup (D'_i \setminus a_{j_i}) \cup \{a_{j_i} = \mathbf{L}_{i+1}\}$
   (d) $D' = D'_n$

   where $j_i \in \{1, \ldots, m\}$ and $0 \leq i \leq n-1$. Then the result follows from point (1).

□

For the $\mathbf{G}$ and $\mathbf{L}_a, \mathbf{L}_b, \mathbf{L}_c$ in Figure 3, we get $[\![\mathbf{G}]\!]_\emptyset = [\![\mathbf{L}_a]\!]_{\{b=\mathbf{L}_b, c=\mathbf{L}_c, d=\mathbf{L}_d\}}$.

6.2. Subject Reduction

Reduction of session environments is standard in session calculi to take into account how communications modify the types of free channels and queues [3, 11].

Definition 29 (Reduction of session environments)**.** According to the reduction rules for processes, we define the reduction rules for session environments:

$$\frac{k \in I}{s[\mathtt{r}_2] : \mathtt{r}_1?\{l_i(S_i).T_i\}_{i\in I}, s : \langle \mathtt{r}_1, \mathtt{r}_2, l_k\langle S_k\rangle\rangle \cdot \mathtt{M} \Rightarrow s[\mathtt{r}_2] : T_k, s : \mathtt{M}}$$

$$\frac{k \in I}{s[\mathtt{r}_1] : \mathtt{r}_2!\{l_i(S_i).T_i\}_{i\in I}, s : \mathtt{M} \Rightarrow s[\mathtt{r}_1] : T_k, s : \mathtt{M} \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l_k\langle S_k\rangle\rangle}$$

$$\frac{\Delta_1 \Rightarrow \Delta_1'}{\Delta_1, \Delta_2 \Rightarrow \Delta_1', \Delta_2}$$

It is easy to verify that reduction preserves the coherence of session environments.

Lemma 1. *If $\Delta \Rightarrow \Delta'$ and Δ is coherent, then Δ' is coherent.*

Theorem 2 (Subject reduction)**.** If $\Gamma \vdash_D P \rhd \Delta$ and $P \longrightarrow^* Q$ and Δ is coherent, then there exists Δ' such that $\Delta \Rightarrow \Delta'$ and $\Gamma \vdash_D Q \rhd \Delta'$.

Proof. The proof is standard by induction on reduction. □

We put the full proof and auxiliary lemmas in Appendix Appendix A.

Subject Reduction Theorem shows the correspondence between processes and local types. So as in [7] we can show that well-typed processes satisfy *session fidelity*, i.e. that exchanged messages follow the order prescribed by global types. The Soundness Theorem gives a relation between the messages of a global type and those of its lightening. A similar relation then holds between the messages in the associated queues.

To formalise this we define that the message queue h' is a *refinement* of the message queue h (notation $h' \ltimes h$) if h' is obtained from h by erasing only messages with empty content. The following two definitions formalise $h' \ltimes h$:

Definition 30. We define $\mathtt{erase}(h)$ a mapping from a queue h to another queue which erases empty-content messages from h:

$$\mathtt{erase}(h) = \begin{cases} \mathtt{erase}(h') & \text{if } h = m \cdot h' \text{ and } m = \langle \mathtt{r}, \mathtt{r}', l\langle\,\rangle\rangle \\ m \cdot \mathtt{erase}(h') & \text{if } h = m \cdot h' \text{ and } m = \langle \mathtt{r}, \mathtt{r}', l\langle v\rangle\rangle \\ h & \text{if } h = \varepsilon \end{cases}$$

Definition 31. We say $h' \ltimes h$ if and only if $h' = \mathtt{erase}(h)$.

To formally state that P will not get stuck during session establishment [link] or [call], we call a group of processes *linkable* such that, in the group, when a process invites others via either [link] or [call] to join a session, this invitation will be successful, i.e. other processes are able to play the roles requested in the invitation:

Definition 32. We say P is *linkable* if one of the following holds:

1. $P_0 = \overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P_0' \in P$ implies that $P_1 \mid ... \mid P_n \in P$ where $P_i = a(y_i[\mathtt{r}_i]).P_i'$ for any $i = 1..n$.
2. $P_0 = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_m, \mathtt{r}_{m+1}..., \mathtt{r}_n\}\rangle.P_0' \in P$ implies that $s : \varepsilon \in P$ and $P_1 \mid ... \mid P_m \mid P_{m+1} \mid ... \mid P_n \in P$ where $P_i = a\langle s[\mathtt{r_i}]\rangle.P_i'$ for any $i = 1..m$ and $P_j = a(y_j[\mathtt{r}_j]).P_j'$ for any $j = (m+1)..n$.

Moreover, for convenience, we define $\longrightarrow^*$ and h/m as follows:

Definition 33. We define $\longrightarrow^* = (\longrightarrow^* \equiv)$ where $\longrightarrow^*$ means reflexive and transitive closure of the relation.

Definition 34. We define / as an operator taking off a message m from h:

$$h/m = \begin{cases} h' & \text{if } h = m \cdot h' \\ \text{Undefined.} & \text{otherwise} \end{cases}$$

Now we state the following theorem which formalises the relation between a $\mathbf{L}$-typed process under D and a $\mathbf{L}'$-typed process under D' where $\mathbb{L}(\mathbf{L}) = (\mathbf{L}', D')$.

Theorem 3. Let $\mathbf{L}$ be a light global type with set of declarations D and $\mathbb{L}(\mathbf{L}) = (\mathbf{L}', D')$ and

$$\Gamma \vdash_D P \rhd \{s[\mathtt{r}] : \mathtt{pj}(\mathbf{L}, \mathtt{r}) \mid \mathtt{r} \in \mathit{role}(\mathbf{L})\} \cup \{s : \varepsilon\}$$

and

$$\Gamma \vdash_{D'} P' \rhd \{s[\mathtt{r}] : \mathtt{pj}(\mathbf{L}', \mathtt{r}) \mid \mathtt{r} \in \mathit{role}(\mathbf{L}')\} \cup \{s : \varepsilon\}.$$

If P and P' are *linkable*, then

1. $P \longrightarrow^* Q \mid s : h$ implies $P' \longrightarrow^* Q' \mid s : h'$ for some Q' and h' such that $h' \ltimes h$;
2. $P' \longrightarrow^* Q' \mid s : h'$ implies $P \longrightarrow^* Q \mid s : h$ for some Q, h such that $h' \ltimes h$.

Proof.

1. When $P \longrightarrow^* Q \mid s : h$. Without loss of generality, assume $P = Q_1 \mid s : \varepsilon$ and $P' = Q_1' \mid s : \varepsilon$.

   By Definition 27 and Theorem 1 (Soundness), when $h_0' \ltimes h_0$:
   (a) If by [bra] $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0/m$ and m's content is not empty and h_0/m is defined, then $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0'/m$ and we have $h_0'/m \ltimes h_0/m$.

(b) If by [bra] $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0/m$ and m's content is empty and h_0/m is defined, then $Q_1' \mid s : h_0' \equiv Q_2' \mid s : h_0'$ and we have $h_0' \ltimes h_0/m$.

(c) If by [sel] $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0 \cdot m$ and m's content is not empty, then $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0' \cdot m$ and we have $h_0' \cdot m \ltimes h_0 \cdot m$.

(d) If by [sel] $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0 \cdot m$ and m's content is empty, then $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0'$ or $Q_1' \mid s : h_0' \equiv Q_2' \mid s : h_0'$ and we have $h_0' \ltimes h_0 \cdot m$.

(e) If by [call] $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0$ such that $h_0 = \varepsilon$ and $P_0 = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.Q_0 \in Q_1$, then we have $a = \mathbf{L}_a \in D$ and $a = \mathbf{L}_a' \in D'$ where $\mathbb{L}(\mathbf{L}_a) = (\mathbf{L}_a', D')$, and thus we have $P_0' = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_{n'}\}\rangle.Q_0' \in Q_1'$ and $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0'$ where $h_0' = \varepsilon$ since P' is *linkable*.

After applying (a) - (e) repeatedly, we have $P' \longrightarrow^* Q' \mid s : h'$ for some Q' and h' such that $h' \ltimes h$.

2. When $P' \longrightarrow^* Q' \mid s : h'$. Without loss of generality, assume $P' = Q_1' \mid s : \varepsilon$ and $P = Q_1 \mid s : \varepsilon$. By Definition 27 and Theorem 1 (Soundness), when $h_0' \ltimes h_0$:

(f) If by [bra] $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0'/m$ and h_0/m is defined, then $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0/m$ and we have $h_0'/m \ltimes h_0/m$.

(g) If by [sel] $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0' \cdot m$, then $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0 \cdot m$ and we have $h_0' \cdot m \ltimes h_0 \cdot m$.

(h) If by [call] $Q_1' \mid s : h_0' \longrightarrow Q_2' \mid s : h_0'$ such that $h_0' = \varepsilon$ and $P_0' = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_{n'}\}\rangle.Q_0' \in Q_1'$, then we have $a = \mathbf{L}_a' \in D'$, which implies that $a = \mathbf{L}_a \in D$ where $\mathbb{L}(\mathbf{L}_a) = (\mathbf{L}_a', D')$, and thus we have $P_0 = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.Q_0 \in Q_1$ and $Q_1 \mid s : h_0 \longrightarrow Q_2 \mid s : h_0$ where $h_0 = \varepsilon$ since P is *linkable*.

After applying (f) - (h) repeatedly, we have $P \longrightarrow^* Q \mid s : h$ for some Q and h such that $h' \ltimes h$.

□

7. Related Works and Conclusion

In the recent literature global types have been enriched in various directions by making them more expressive through roles [5, 12] or logical assertions [13] or states [14] or monitoring [7], and by making them safer through security levels for data and participants [15, 16] or reputation systems [17].

Inspired by conversation calculus [18, 19], where roles can dynamically join and leave, our processes have prefixes inviting roles to join an existing session or accepting to join a new session. In [20] roles are defined as classes of local

behaviours that an arbitrary and changing number of participants can incarnate. Recently, the work [21] proposes a typed framework for the analysis of multiparty communications with dynamic role authorisation and delegation.

The more related works are [22, 23, 24]. In [22], Demangeon and Honda introduce nested protocols, that is, the possibility to define a subprotocol independently of its parent protocol, which calls the subprotocol explicitly. Through a call, arguments can be passed, such as values, roles and other protocols, allowing higher-order description. Therefore global types in [22] are much more expressive than our light types. Carbone and Montesi [23] propose to merge together protocols interleaved in the same choreography into a single global type, removing costly invitations. Their approach is opposite to ours, and they deal with implementations, while we deal with types. Caires and Pérez [24] develop formal relationships between multiparty and binary global types: they decompose multiparty global types into binary global types using typed *medium processes*. Therefore the goal and the results of [24] differ from ours.

We do not consider session delegation, that needs a careful treatment. In fact delegation requires two sessions, one in which the delegated channel is active and the other in which this channel is exchanged. Therefore lightening the global type of the first session can require to modify the type of the delegated channel in the second session. We leave this problem as future work.

The novelty of this paper is a method for lightening (i.e. decomposing) complicated global types by removing redundant interactions and preserving the order of the remaining interactions. This method produces modularised global types which are invoked by the calls constructor and together preserve the meaning of the original type. Moreover, we prove the soundness of the lightening method and we provide a simple type system enforcing sound behaviours of endpoint processes and enjoying the property of subject reduction.

Acknowledgements. This work is sponsored by Torino University/Compagnia San Paolo Project SALT. We thank Prof. Mariangiola Dezani-Ciancaglini for her valuable suggestions and discussions. We also thank the reviewers of PLACES 2014 and the reviewers of JLAMP (Journal of Logical and Algebraic Meth. in Progr.) for careful reading, since we deeply revised this article following their suggestions.

References

[1] T.-C. Chen, Lightening global types, in: PLACES, Vol. 155 of EPTCS, 2014, pp. 38–46.

[2] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, N. Yoshida, Scribbling interactions with a formal foundation, in: ICDCIT, Vol. 6536 of LNCS, 2011, pp. 55–75.

[3] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM, 2008, pp. 273–284.

[4] L. Bettini, M. Coppo, L. DAntoni, M. De Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: CONCUR, Vol. 5201 of LNCS, Springer, 2008, pp. 418–433.

[5] P.-M. Deniélou, N. Yoshida, Dynamic multirole session types, in: POPL, ACM, 2011, pp. 435–446.

[6] P.-M. Deniélou, N. Yoshida, Multiparty session types meet communicating automata, in: ESOP, Vol. 7211 of LNCS, Springer, 2012, pp. 194–213.

[7] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, N. Yoshida, Monitoring networks through multiparty session types, in: FMOODS/FORTE, Vol. 7892 of LNCS, Springer, 2013, pp. 50–65.

[8] D. Mostrous, N. Yoshida, Session-based communication optimisation for higher-order mobile processes, in: TLCA, Vol. 5608 of LNCS, Springer, 2009, pp. 203–218.

[9] D. Mostrous, Session types in concurrent calculi: Higher-order processes and objects, Ph.D. thesis, Imperial College London (2009).

[10] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party sessions, Log. Meth. Comp. Scie. 8 (2012) 1–45.

[11] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP, Vol. 1381 of LNCS, Springer, 1998, pp. 122–138.

[12] P. Baltazar, L. Caires, V. T. Vasconcelos, H. T. Vieira, A type system for flexible role assignment in multiparty communicating systems, in: TGC, Vol. 8191 of LNCS, Springer, 2013, pp. 82–96.

[13] L. Bocchi, K. Honda, E. Tuosto, N. Yoshida, A theory of design-by-contract for distributed multiparty interactions, in: CONCUR, Vol. 6269 of LNCS, Springer, 2010, pp. 162–176.

[14] T.-C. Chen, K. Honda, Specifying stateful asynchronous properties for distributed programs, in: CONCUR, Vol. 7454 of LNCS, 2012, pp. 209–224.

[15] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, T. Rezk, Session types for access and information flow control, in: CONCUR, Vol. 6269 of LNCS, Springer, 2010, pp. 237–252.

[16] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, Information flow safety in multiparty sessions, in: EXPRESS, Vol. 64 of EPTCS, 2011, pp. 16–31.

[17] V. Bono, S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, A reputation system for multirole sessions, in: TGC, Vol. 7173 of LNCS, Springer, 2012, pp. 1–24.

[18] H. T. Vieira, L. Caires, J. C. Seco, The conversation calculus: A model of service-oriented computation, in: ESOP, Vol. 4960, Springer, 2008, pp. 269–283.

[19] L. Caires, H. T. Vieira, Conversation types, Theor. Comput. Sci. 411 (51-52) (2010) 4399–4440.

[20] P. Deniélou, N. Yoshida, Dynamic multirole session types, in: POPL, ACM, 2011, pp. 435–446.

[21] S. Ghilezan, S. Jaksic, J. Pantovic, J. A. Pérez, H. T. Vieira, Dynamic role authorization in multiparty conversations, in: BEAT, Vol. 162 of EPTCS, 2014, pp. 1–8.

[22] R. Demangeon, K. Honda, Nested protocols in session types, in: CONCUR, Vol. 7454 of LNCS, Springer, 2012, pp. 272–286.

[23] M. Carbone, F. Montesi, Merging multiparty protocols in multiparty choreographies, in: PLACES, Vol. 109 of EPTCS, 2012, pp. 21–27.

[24] L. Caires, J. A. Pérez, A typeful characterization of multiparty structured conversations based on binary sessions, CoRR abs/1407.4242.

Appendix A. Full Proof and Auxiliary Lemmas for Theorem 2

As usual the proof of subject reduction relies on an inversion lemma and a substation lemma.

Lemma 2 (Inversion lemma)**.**

1. If $\Gamma \vdash_D \mathbf{0} \triangleright \Delta$, then Δ is end only.
2. If $\Gamma \vdash_D \overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P \triangleright \Delta$, then $\Gamma \vdash_D P \triangleright \Delta, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})$ and $\{\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_n\} = \mathit{role}(D(a))$.
3. If $\Gamma \vdash_D \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P \triangleright \Delta$, then $\Delta = \Delta', s[\mathtt{r}] : \mathsf{inv}\langle a\rangle$ and $\Gamma \vdash_D P \triangleright \Delta', s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})$ and $\{\mathtt{r}_1, ..., \mathtt{r}_n\} = \mathit{role}(D(a))$.
4. If $\Gamma \vdash_D a(y[\mathtt{r}]).P \triangleright \Delta$, then $\Gamma \vdash_D P \triangleright \Delta, y[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})$.
5. If $\Gamma \vdash_D a\langle s[\mathtt{r}]\rangle.P \triangleright \Delta$, then $\Delta = \Delta', s[\mathtt{r}] : \mathsf{acc}\langle a\rangle$ and $\Gamma \vdash_D P \triangleright \Delta', s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r})$.
6. If $\Gamma \vdash_D u[\mathtt{r}]!\langle \mathtt{r}', l\langle e\rangle\rangle.P \triangleright \Delta$, then $\Delta = \Delta', u[\mathtt{r}] : \mathtt{r}'!\{l_i(S_i).T_i\}_{i\in I}$ and $l = l_k$ and $\Gamma \vdash e : S_k$ and $\Gamma \vdash_D P \triangleright \Delta', u[\mathtt{r}] : T_k$ and $k \in I$.
7. If $\Gamma \vdash_D u[\mathtt{r}]?\langle \mathtt{r}', \{l_i(x_i).P_i\}_{i\in I}\rangle \triangleright \Delta$, then $\Delta = \Delta', u[\mathtt{r}] : \mathtt{r}'?\{l_i(S_i).T_i\}_{i\in I}$ and $\Gamma, x : S_i \vdash_D P \triangleright \Delta', u[\mathtt{r}] : T_i$ for all $i \in I$.
8. If $\Gamma \vdash_D \mathit{if}\ e\ \mathit{then}\ P\ \mathit{else}\ Q \triangleright \Delta$, then $\Gamma \vdash e : \mathsf{Bool}$ and $\Gamma \vdash_D P \triangleright \Delta$ and $\Gamma \vdash_D Q \triangleright \Delta$.
9. If $\Gamma \vdash_D P_1 \mid P_2 \triangleright \Delta$, then $\Delta = \Delta_1, \Delta_2$ and $\Gamma \vdash_D P_1 \triangleright \Delta_1$ and $\Gamma \vdash_D P_2 \triangleright \Delta_2$.
10. If $\Gamma \vdash_D s : \varepsilon \triangleright \Delta$, then $\Delta = \{s : \epsilon\}$.
11. If $\Gamma \vdash_D s : h \cdot \langle \mathtt{r}, \mathtt{r}', l\langle v\rangle\rangle \triangleright \Delta$, then $\Delta = \Delta', s : \mathtt{M} \cdot \langle \mathtt{r}, \mathtt{r}', l\langle S\rangle\rangle$ and $\Gamma \vdash v : S$ and $\Gamma \vdash_D s : h \triangleright \Delta', s : \mathtt{M}$.
12. If $\Gamma \vdash_D (\nu s)P \triangleright \Delta$, then $\Delta = \Delta' \setminus s$ and $\Gamma \vdash_D P \triangleright \Delta'$ and Δ' is coherent for D and s.

Proof. By induction on derivations. □

Lemma 3 (Substitution lemma)**.**

1. If $\Gamma, x : S \vdash_D P \triangleright \Delta$ and $\Gamma \vdash v : S$, then $\Gamma \vdash_D P\{v/x\} \triangleright \Delta$.
2. If $\Gamma \vdash_D P \triangleright \Delta, y[\mathtt{r}] : T$ and $s \notin \mathit{dom}(\Delta)$, then $\Gamma \vdash_D P\{s/y\} \triangleright \Delta, s[\mathtt{r}] : T$.

Proof. Standard. □

A last lemma deals with the types of queues.

Lemma 4 (Types of queues)**.** If $\Gamma \vdash_D s : \langle \mathtt{r}, \mathtt{r}', l\langle v\rangle\rangle \cdot h \rhd \Delta$, then

$$\Delta = \Delta', s : \langle \mathtt{r}, \mathtt{r}', l\langle S\rangle\rangle \cdot \mathtt{M} \text{ and } \Gamma \vdash v : S \text{ and } \Gamma \vdash_D s : h \rhd \Delta', s : \mathtt{M}.$$

Proof. By induction on n we show that, if

$$\Gamma \vdash_D s : \langle \mathtt{r}_1, \mathtt{r}'_1, \dagger_1\rangle \cdot \ldots \cdot \langle \mathtt{r}_n, \mathtt{r}'_n, \dagger_n\rangle \rhd \Delta$$

then

$$\Delta = \Delta', s : \langle \mathtt{r}_1, \mathtt{r}'_1, \ddagger_1 \cdot \ldots \cdot \langle \mathtt{r}_n, \mathtt{r}'_n, \ddagger_n\rangle\rangle$$

and $\Gamma \vdash v_j : S_j$ and $\dagger_j = l_j\langle v_j\rangle$ for $j \in J$. The first step follows from Lemma 2.10. The induction step follows from Lemma 2.11. □

Theorem 4 (Invariance of types by structural congruence)**.** If $\Gamma \vdash_D P \rhd \Delta$ and $P \equiv Q$, then $\Gamma \vdash_D Q \rhd \Delta$.

Proof. The proof is by induction on $\equiv$ defined in Figure 6. We only consider the case $s : h \cdot m_1 \cdot m_2 \cdot h' \equiv s : h \cdot m_2 \cdot m_1 \cdot h'$ since $m_1 \cdot m_2 \curvearrowright m_2 \cdot m_1$. By the proof of Lemma 4

$$\Gamma \vdash_D s : h \cdot m_1 \cdot m_2 \cdot h' \rhd \Delta$$

implies $\Delta = \Delta', s : \mathtt{M}_0 \cdot \mathtt{M}_1 \cdot \mathtt{M}_2 \cdot \mathtt{M}_3$ and $\Gamma \vdash_D s : h \rhd \Delta', s : \mathtt{M}_0$, $\Gamma \vdash_D s : m_1 \rhd \Delta', s : \mathtt{M}_1$, $\Gamma \vdash_D s : m_2 \rhd \Delta', s : \mathtt{M}_2$ and $\Gamma \vdash_D s : h' \rhd \Delta', s : \mathtt{M}_3$. It is easy to verify that

$$\Gamma \vdash_D s : h \cdot m_2 \cdot m_1 \cdot h' \rhd \Delta', s : \mathtt{M}_0 \cdot \mathtt{M}_2 \cdot \mathtt{M}_1 \cdot \mathtt{M}_3$$

Since $m_1 \cdot m_2 \curvearrowright m_2 \cdot m_1$ implies $\mathtt{M}_2 \cdot \mathtt{M}_1 \curvearrowright \mathtt{M}_1 \cdot \mathtt{M}_2$ we derive

$$\Gamma \vdash_D s : h \cdot m_2 \cdot m_1 \cdot h' \rhd \Delta', s : \mathtt{M}_0 \cdot \mathtt{M}_1 \cdot \mathtt{M}_2 \cdot \mathtt{M}_3$$

by rule [T-equiv]. □

Theorem 2 : Subject reduction . If $\Gamma \vdash_D P \rhd \Delta$ and $P \longrightarrow^* Q$ and Δ is coherent, then there exists Δ' such that $\Delta \Rightarrow \Delta'$ and $\Gamma \vdash_D Q \rhd \Delta'$.

Proof. By induction on the reduction rules and by cases on the last applied rule.

1. Rule [Link]. In this case $P = \overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P' \mid a(y_1[\mathtt{r}_1]).P_1 \mid ... \mid a(y_n[\mathtt{r}_n]).P_n$ and

$$\Gamma \vdash_D P \rhd \Delta. \tag{A.1}$$

By applying Lemma 2.9 to (A.1), we get

$$\Gamma \vdash_D \overline{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_n\}\rangle.P' \rhd \Delta_1 \tag{A.2}$$

$$\Gamma \vdash_D a(y_1[\mathtt{r}_1]).P_1 \mid ... \mid a(y_n[\mathtt{r}_n]).P_n \rhd \Delta_2 \tag{A.3}$$

where $\Delta = \Delta_1, \Delta_2$. By applying Lemma 2.2 to (A.2), we have

$$\Gamma \vdash_D P' \rhd \Delta_1, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r}) \tag{A.4}$$

$$\{\mathtt{r}, \mathtt{r}_1 ... \mathtt{r}_n\} = \mathit{role}(D(a)) \tag{A.5}$$

By applying Lemma 2.9 n times to (A.3), we have for all $i \in \{1, \ldots, n\}$:

$$\Gamma \vdash_D a(y_i[\mathtt{r}_i]).P_i \rhd \Delta_i' \tag{A.6}$$

where $\bigcup_{1 \leq i \leq n} \Delta_i' = \Delta_2$. By applying Lemma 2.4 to (A.6), we have for all $i \in \{1, \ldots, n\}$:

$$\Gamma \vdash_D P_i \rhd \Delta_i', y_i[\mathtt{r}_i] : \mathtt{pj}(D(a), \mathtt{r}_i) \tag{A.7}$$

so that $\Delta_2 = \bigcup_{1 \leq i \leq n} \Delta_i'$. By applying Lemma 3 to (A.7), we have for all $i \in \{1, \ldots, n\}$:

$$\Gamma \vdash_D P_i\{s/y_i\} \rhd \Delta_i', s[\mathtt{r}_i] : \mathtt{pj}(D(a), \mathtt{r}_i) \tag{A.8}$$

By applying rule [T-par] $n+1$ times to (A.4) and (A.8), we derive

$$\Gamma \vdash_D P' \mid P_1\{s/y_1\} \mid ... \mid P_n\{s/y_n\} \rhd \Delta_1, \Delta_2, \bigcup_{\mathtt{r}' \in \mathit{role}(D(a))} \{s[\mathtt{r}'] : \mathtt{pj}(D(a), \mathtt{r}')\} \tag{A.9}$$

By rule [T-qnull], we derive

$$\Gamma \vdash_D s : \varepsilon \rhd \{s : \epsilon\} \tag{A.10}$$

By applying rule [T-par] to (A.9) and (A.10), we derive

$$\Gamma \vdash_D P' \mid P_1\{s/y_1\} \mid ... \mid P_n\{s/y_n\} \mid s : \varepsilon \rhd \Delta_1, \Delta_2, \bigcup_{\mathtt{r}' \in \mathit{role}(D(a))} \{s[\mathtt{r}'] : \mathtt{pj}(D(a), \mathtt{r}')\} \cup \{s : \epsilon\} \tag{A.11}$$

Let $\mathtt{pj}(D(a), \mathtt{r}') = T_{\mathtt{r}'}$ for $\mathtt{r}' \in \mathit{role}(D(a))$. The remainders $T_{\mathtt{r}'} - \epsilon \upharpoonright \mathtt{r}' = T_{\mathtt{r}'}$ are defined for all $\mathtt{r}' \in \mathit{role}(D(a))$. Moreover $\mathtt{pjl}(T_{\mathtt{r}'}, \mathtt{r}'') \bowtie \mathtt{pjl}(T_{\mathtt{r}''}, \mathtt{r}')$ for all $\mathtt{r}', \mathtt{r}'' \in \mathit{role}(D(a))$ with $\mathtt{r}' \neq \mathtt{r}''$ by Proposition 2. Thus $\Delta_1, \Delta_2, \bigcup_{\mathtt{r}' \in \mathit{role}(D(a))} \{s[\mathtt{r}'] : \mathtt{pj}(D(a), \mathtt{r}')\} \cup \{s : \epsilon\}$ is coherent since Δ_1, Δ_2 is coherent. Then we can apply rule [T-news] to (A.11) and derive

$$\Gamma \vdash_D (\nu s)(P' \mid P_1\{s/y_1\} \mid ... \mid P_n\{s/y_n\} \mid s : \varepsilon) \rhd \Delta_1, \Delta_2 \tag{A.12}$$

2. Rule [call]. In this case $P = (\nu s)(P_0) \mid a(y_1[\mathtt{r}_{m+1}]).P_{m+1} \mid ... \mid a(y_{n-m}[\mathtt{r}_n]).P_n$ where

$$P_0 = \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_m, \mathtt{r}_{m+1} ..., \mathtt{r}_n\}\rangle.P' \mid a\langle s[\mathtt{r}_1]\rangle.P_1 \mid ... \mid a\langle s[\mathtt{r}_m]\rangle.P_m \mid s : \varepsilon$$

and

$$\Gamma \vdash_D P \rhd \Delta \tag{A.13}$$

By applying Lemma 2.9 to (A.13), we have

$$\Gamma \vdash_D (\nu s)P_0 \rhd \Delta_1 \tag{A.14}$$

$$\Gamma \vdash_D a(y_1[\mathtt{r}_{m+1}]).P_{m+1} \mid ... \mid a(y_{n-m}[\mathtt{r}_n]).P_n \rhd \Delta_2 \tag{A.15}$$

where $\Delta = \Delta_1, \Delta_2$. By applying Lemma 2.12 to (A.14), we have

$$\Gamma \vdash_D P_0 \rhd \Delta_0 \tag{A.16}$$

and $\Delta_1 = \Delta_0 \setminus s$ and Δ_0 is coherent for D and s. By applying Lemma 2.9 $m+2$ times to (A.16), we have

$$\Gamma \vdash_D \widehat{a}\langle s, \{\mathtt{r}_1, ..., \mathtt{r}_m, \mathtt{r}_{m+1}, ..., \mathtt{r}_n\}\rangle.P' \rhd \Delta'_0 \tag{A.17}$$

$$\Gamma \vdash_D a\langle s[\mathtt{r}_i]\rangle.P_i \rhd \Delta'_i \quad \text{for } 1 \leq i \leq m \tag{A.18}$$

$$\Gamma \vdash_D s : \varepsilon \rhd \{s : \epsilon\} \tag{A.19}$$

where $\bigcup_{0 \leq i \leq m} \Delta'_i \cup \{s : \epsilon\} = \Delta_0$. Similarly, by applying Lemma 2.9 $n-m$ times to (A.15), we have

$$\Gamma \vdash_D a(y_i[\mathtt{r}_{m+j}]).P_{m+j} \rhd \Delta'_{m+j} \quad \text{for } 1 \leq j \leq n-m \tag{A.20}$$

where $\bigcup_{1 \leq j \leq n-m} \Delta'_{m+j} = \Delta_2$. By applying Lemma 2.3 to (A.17), we have $\Delta'_0 = \Delta''_0, s[\mathtt{r}] : \mathsf{inv}\langle a\rangle$ and

$$\Gamma \vdash_D P' \rhd \Delta''_0, s[\mathtt{r}] : \mathtt{pj}(D(a), \mathtt{r}) \tag{A.21}$$

$$\{\mathtt{r}, \mathtt{r}_1, ..., \mathtt{r}_n\} = \mathit{role}(D(a)) \tag{A.22}$$

By applying Lemma 2.5 to (A.18), we have $\Delta'_i = \Delta''_i, s[\mathtt{r}_i] : \mathsf{acc}\langle a\rangle$ for $1 \leq i \leq m$ and

$$\Gamma \vdash_D P_i \rhd \Delta''_i, s[\mathtt{r}_i] : \mathtt{pj}(D(a), \mathtt{r}_i) \quad \text{for } 1 \leq i \leq m \tag{A.23}$$

By applying Lemma 2.4 to (A.20), we have for $1 \leq j \leq n-m$

$$\Gamma \vdash_D P_{m+j} \rhd \Delta'_{m+j}, y_j[\mathtt{r}_{m+j}] : \mathtt{pj}(D(a), \mathtt{r}_{m+j}) \tag{A.24}$$

By applying Lemma 3 to (A.24), we have for $1 \leq j \leq n-m$

$$\Gamma \vdash_D P_{m+j}\{s/y_j\} \rhd \Delta'_{m+j}, s[\mathtt{r}_{m+j}] : \mathtt{pj}(D(a), \mathtt{r}_{m+j}) \tag{A.25}$$

By applying [T-par] $n+2$ times to (A.19), (A.21), (A.23), and (A.25), taking into account (A.22), we derive

$$\Gamma \vdash_D P' \mid P_1 \mid ... \mid P_m \mid P_{m+1}\{s/y_1\} \mid ... \mid P_n\{s/y_{n-m}\} \mid s : \varepsilon \rhd \Delta_1, \Delta_2, \bigcup_{\mathtt{r}' \in \mathit{role}(D(a))} \{s[\mathtt{r}'] : \mathtt{pj}(D(a), \mathtt{r}')\} \cup \{s : \epsilon\} \tag{A.26}$$

since $\Delta_1 = \bigcup_{0 \le i \le m} \Delta_i''$ due to $\Delta_1 = \Delta_0 \setminus s$. Reasoning as the case of rule [link], we can show that

$$\Delta_1, \Delta_2, \bigcup_{\mathtt{r}' \in role(D(a))} \{s[\mathtt{r}'] : \mathtt{pj}(D(a), \mathtt{r}')\} \cup \{s : \epsilon\}$$

is coherent. Therefore we can apply rule [T-news] to (A.26) and derive

$$\Gamma \vdash_D (\nu s)(P' \mid P_1 \mid ... \mid P_m \mid P_{m+1}\{s/y_1\} \mid ... \mid P_n\{s/y_{n-m}\} \mid s : \varepsilon) \rhd \Delta$$

3. Rule [sel]. In this case $P = s[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle e \rangle\rangle.P' \mid s : h$ and $e \downarrow v$ and

$$\Gamma \vdash_D P \rhd \Delta. \tag{A.27}$$

By applying Lemma 2.9 to (A.27), we get

$$\Gamma \vdash_D s[\mathtt{r}_1]!\langle \mathtt{r}_2, l\langle v \rangle\rangle.P' \rhd \Delta_1 \tag{A.28}$$
$$\Gamma \vdash_D s : h \rhd \Delta_2, \tag{A.29}$$

where $\Delta = \Delta_1, \Delta_2$. By applying Lemma 2.6 to (A.28), we have

$$\Delta_1 = \Delta_1', s[\mathtt{r}_1] : \mathtt{r}_2!\{l_i(S_i).T_i\}_{i \in I} \tag{A.30}$$
$$l = l_k$$
$$\Gamma \vdash_D P' \rhd \Delta_1', s[\mathtt{r}_1] : T_k \tag{A.31}$$
$$\Gamma \vdash e : S_k \tag{A.32}$$

and $k \in I$. By (A.32) and $e \downarrow v$, we have

$$\Gamma \vdash v : S_k \tag{A.33}$$

By applying Lemma 2.10 and 2.11 to (A.29), we have $\Delta_2 = \Delta_2', s : \mathtt{M}$. By applying rule [T-q] to (A.29) and (A.33), we derive

$$\Gamma \vdash_D s : h \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle v \rangle\rangle \rhd \Delta_2', s : \mathtt{M} \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle S_k \rangle\rangle \tag{A.34}$$

By applying rule [T-par] to (A.31) and (A.34), we derive

$$\Gamma \vdash_D s : h \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle v \rangle\rangle \mid P' \rhd \Delta_1', \Delta_2', s : \mathtt{M} \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle S_k \rangle\rangle, s[\mathtt{r}_1] : T_k$$

By Definition 29,

$$\Delta \Rightarrow \Delta_1', \Delta_2', s : \mathtt{M} \cdot \langle \mathtt{r}_1, \mathtt{r}_2, l\langle S_k \rangle\rangle, s[\mathtt{r}_1] : T_k$$

and this concludes the proof.

4. Rule [bra]. In this case $P = s : \langle \mathtt{r}_2, \mathtt{r}_1, l_k\langle v \rangle\rangle \cdot h \mid s[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i'\}_{i \in I}\rangle$ and $k \in I$ and

$$\Gamma \vdash_D P \rhd \Delta. \tag{A.35}$$

By applying Lemma 2.9 to (A.35), we get

$$\Gamma \vdash_D s : \langle \mathtt{r}_2, \mathtt{r}_1, l\langle v\rangle\rangle \cdot h \rhd \Delta_1, \tag{A.36}$$

$$\Gamma \vdash_D s[\mathtt{r}_1]?\langle \mathtt{r}_2, \{l_i(x_i).P_i'\}_{i\in I}\rangle \rhd \Delta_2 \tag{A.37}$$

where $\Delta = \Delta_1, \Delta_2$. By applying Lemma 4 to (A.36), we get

$$\Gamma \vdash_D s : h \rhd \Delta_1', s : \mathtt{M} \tag{A.38}$$

$$\Delta_1 = \Delta_1', s : \langle \mathtt{r}_2, \mathtt{r}_1, l_k\langle S\rangle\rangle \cdot \mathtt{M}$$

$$\Gamma \vdash v : S \tag{A.39}$$

By applying Lemma 2.7 to (A.37), we have

$$\Delta_2 = \Delta_2', s[\mathtt{r}_1] : \mathtt{r}_2?\{l_i(S_i).T_i\}_{i\in I}$$

$$\forall i \in I : \Gamma, x_i : S_i \vdash_D P_i' \rhd \Delta_2', s[\mathtt{r}_1] : T_i \tag{A.40}$$

The coherence of Δ implies $S = S_k$. By applying Lemma 3 to (A.40) and (A.39) we get

$$\Gamma \vdash_D P_k'\{v/x_k\} \rhd \Delta_2', s[\mathtt{r}_1] : T_k \tag{A.41}$$

By applying rule [T-par] to (A.38) and (A.41), we derive

$$\Gamma \vdash_D s : h \mid P_k'\{v/x_k\} \rhd \Delta_1', \Delta_2', s : \mathtt{M}, s[\mathtt{r}_1] : T_k.$$

By Definition 29,

$$\Delta \Rightarrow \Delta_1', \Delta_2', s : \mathtt{M}, s[\mathtt{r}_1] : T_k$$

and this concludes the proof.

□