Lightening Global Types

Tzu-chun Chen

Dipartimento di Informatica, Università di Torino, Italy

`chen@di.unito.it`

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. Lightening global types gives us cleaner global types, which keep all necessary communications. 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.

1 Introduction. Since cooperating tasks and sharing resources through communications under network infrastructures (e.g. clouds, large-scale distributed systems, etc.) has 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 applications of interactions. For this aim, Scribble [19], a communication-based programming language, is introduced building on the theory of global types [21, 3]. A developer can use Scribble to code a global protocol, which stipulates any local endpoints (i.e. local applications) participanting in it. The merits of coding global protocols, rather than just coding the local ones, are (1) giving all local participants a clear blue map of what events they are involved in and what purposes of those events and (2) making it easier and more efficient to exchange, share, and maintain communications plans (e.g. design of global protocols) across organisations. However, the tool itself cannot ensure an efficient communication programming. The scenario of a global communication can be very complicated so it becomes a burden for programmers to correctly code interactions which satisfy protocols (described by global types). At runtime, the cost for keeping all resources ready for a long communication and for maintaining the safety of the whole system can increase a lot.

For example, assume a gift requester needs a *key* (with her identity) to get a wanted gift. In order to get the key, she needs to get a *guide*, which is a map for finding the key. It is like searching for treasures step by step, where a player needs not be always online in *one* session for completing the whole procedure. Instead, the communication protocol can be viewed as *separated but related* sessions which are linked (we use **call**s to switch from a session to another). For example, let a session $\mathscr{A}$ do the interactions for getting *guide*, and another session $\mathscr{B}$ do the interactions for getting *key* with *guide*. Both *guide* and *key* are knowledge gained from these interactions. *guide* bridges $\mathscr{A}$ and $\mathscr{B}$ as it is gained in $\mathscr{A}$ and used in $\mathscr{B}$. The participant then uses *key*, if she successfully got it in $\mathscr{B}$, to gain the wanted gift. Let session $\mathscr{C}$ implements these final interactions.

As standard we use global types [21, 3] to describe interaction protocols, adding a **call** command and type declarations for relating sessions. We call *light global types* the global types written in the extended syntax. Figure 1 simply represents the difference between viewing all interactions as one scenario and viewing interactions as three separated ones. As usual $\mathtt{store} \to \mathtt{req} : \{\mathit{yes3}(\mathtt{str}).\mathtt{end}, \mathit{no3}().\mathtt{end}\}$ models a communication where role $\mathtt{store}$ sends to role $\mathtt{req}$ either the label $\mathit{yes3}$ and a value of type $\mathtt{str}$ or the label $\mathit{no3}$, and in both cases $\mathtt{end}$ finishes the interaction. The construct **call** ℓ_b indicates that the interaction should continue by executing the session described by the light global type associated to the name ℓ_b.

$$
\begin{aligned}
G \;=\; &\texttt{req} \rightarrow \texttt{map} : \textit{req1}(\texttt{str}). \\
&\texttt{map} \rightarrow \texttt{req} : \\
&\{\, \textit{yes1}(\texttt{str}). \\
&\quad \texttt{req} \rightarrow \texttt{issuer} : \textit{req2}(\texttt{str}). \\
&\quad \texttt{issuer} \rightarrow \texttt{req} : \\
&\quad \{\, \textit{yes2}(\texttt{int}). \\
&\qquad \texttt{req} \rightarrow \texttt{store} : \textit{req3}(\texttt{int}). \\
&\qquad \texttt{store} \rightarrow \texttt{req} : \\
&\qquad \{\, \textit{yes3}(\texttt{str}).\texttt{end}, \textit{no3}().\texttt{end} \,\}, \\
&\qquad \textit{no2}().\texttt{req} \rightarrow \texttt{store} : \textit{no4}().\texttt{end} \,\}, \\
&\quad \textit{no1}(). \\
&\quad \texttt{req} \rightarrow \texttt{issuer} : \textit{no5}(). \\
&\quad \texttt{req} \rightarrow \texttt{store} : \textit{no6}().\texttt{end} \,\}
\end{aligned}
$$

$$
\begin{aligned}
\ell_a \;=\; &\texttt{req} \rightarrow \texttt{map} : \textit{req1}(\texttt{str}). \\
&\texttt{map} \rightarrow \texttt{req} : \\
&\{\, \textit{yes1}(\texttt{str}).\textbf{call}\ \ell_b, \textit{no1}().\texttt{end} \,\}
\end{aligned}
$$

$$
\begin{aligned}
\ell_b \;=\; &\texttt{req} \rightarrow \texttt{issuer} : \textit{req2}(\texttt{str}). \\
&\texttt{issuer} \rightarrow \texttt{req} : \\
&\{\, \textit{yes2}(\texttt{int}).\textbf{call}\ \ell_c, \textit{no2}().\texttt{end} \,\}
\end{aligned}
$$

$$
\begin{aligned}
\ell_c \;=\; &\texttt{req} \rightarrow \texttt{store} : \textit{req3}(\texttt{int}). \\
&\texttt{store} \rightarrow \texttt{req} : \\
&\{\, \textit{yes3}(\texttt{str}).\texttt{end}, \textit{no3}().\texttt{end} \,\}
\end{aligned}
$$

Figure 1: Viewing interactions as a whole or as separated but related ones.

The set of light global types associated to the names ℓ_a, ℓ_b and ℓ_c describe the same protocol given by the global type G. An advantage of lightening is to spare communications needed only to warn participants they will not receive further messages. We call such communications *redundant* ones. In the example we avoid the interactions $\texttt{req} \rightarrow \texttt{store} : \textit{no4}()$ and $\texttt{req} \rightarrow \texttt{issuer} : \textit{no5}().\texttt{req} \rightarrow \texttt{store} : \textit{no6}()$. Moreover, lightening prevents both local participants and global network from wasting resources, e.g. keeping online or waiting for step-by-step permissions.

Features of lightening become more clear by looking at the Scribble code implementing the global types of Figure 1, see Figure 2. The words in bold are keywords. In Scribble after the **protocol** keyword one writes the protocol's name and declares the roles involved in the session, then one describes the interactions in the body. The left-hand side (LHS) of Figure 2 shows the Scribble code corresponding to the global type G. Although the code is for a simple task, it is involved since there are three nested **choice**s. On the contrary, the right-hand side (RHS) of Figure 2 clearly illustrates the steps for getting a gift in three small protocols (corresponding to the light global types associated to ℓ_a, ℓ_b, ℓ_c respectively), which are separated but linked (by **run** and **at**) for preserving the causality.

The structure of the paper is the following. Section 2 introduces a framework of light global session types, which gives a syntax to easily compose a global type by light global types. Section 3 proposes a function for decomposing a general global type into light global types, e.g. it decomposes G into the types associated to ℓ_a, ℓ_b and ℓ_c. Section 4 proves the soundness of the function, and Section 5 discusses related and future works.

2 Syntax of (light) global types. The syntax for *global types* is standard:

$$
G \;::=\; \texttt{r}_1 \rightarrow \texttt{r}_2 : \{l_j(S_j).G_j\}_{j \in J} \mid \mu t.G \mid t \mid \texttt{end} \quad \text{where } S \;::=\; \texttt{unit} \mid \texttt{nat} \mid \texttt{str} \mid \texttt{bool}
$$

The branching $\texttt{r}_1 \rightarrow \texttt{r}_2 : \{l_j(S_j).G_j\}_{j \in J}$ says that role $\texttt{r}_1$ sends a label l_j and a message of type S_j to $\texttt{r}_2$ by selecting $j \in J$ and then the interaction continues as described in G_j. $\mu t.G$ is a recursive type, where t is guarded in G in the standard way. $\texttt{end}$ means termination of the protocol. We write $l()$ as short for $l(\texttt{unit})$ and we omit brackets when there is only one branch.

```
protocol getGift
        (role req, role map,
         role issuer, role store){
  req1(str identity) from req to map;
  choice at map{
    yes1(str guide) from map to req;
    req2(str guide) from req to issuer;
    choice at issuer{
      yes2(int key) from issuer to req;
      req3(int key) from req to store;
      choice at store{
        yes3(str gift) from store to req;
      } or {
        no3() from store to req;
      }
    } or {
      no2() from issuer to req;
      no4() from req to store;
    }
  } or {
    no1() from map to req;
    no5() from req to issuer;
    no6() from req to store;
  }
}
```

```
protocol getGuide (role req, role map){
  req1(str identity) from req to map;
  choice at map{
    yes1(str guide) from map to req;
    run protocol
          getKey(role req, role issuer) at req;
  } or {
    no1() from map to req;
  }
}
protocol getKey (role req, role issuer){
  req2(str guide) from req to issuer;
  choice at issuer{
    yes2(int key) from issuer to req;
    run protocol
          getGift'(role req, role store) at req;
  } or {
    no2() from issuer to req;
  }
}
protocol getGift' (role req, role store){
  req3(int key) from req to store;
  choice at store{
    yes3(str gift) from store to req;
  } or {
    no3() from store to req;
  }
}
```

Figure 2: Scribble for **protocol**s getGift (LHS) and for getGuide, getKey, and getGift′ (RHS).

Light global types are global types extended with declarations and the construct **call**:

$$D \quad ::= \quad \overrightarrow{\ell = L} \qquad\qquad L \quad ::= \quad \mathtt{r_1} \to \mathtt{r_2} : \{l_j(S_j).L_j\}_{j \in J} \mid \mu t.L \mid G \mid \mathbf{call}\ \ell$$

Declarations associate names to light global types, for example in Figure 1 the name ℓ_a is associated with the type

$$\mathtt{req} \to \mathtt{map} : \mathit{req1}(\mathtt{str}).\ \mathtt{map} \to \mathtt{req} : \{\ \mathit{yes1}(\mathtt{str}).\mathbf{call}\ \ell_b, \mathit{no1}().\mathtt{end}\ \}$$

We denote by $\emptyset$ the empty declaration.

The type **call** ℓ prescribes that the interaction continues by opening a new session with light global type L such that $\ell = L$ belongs to the set of current declarations. For example according to the declarations in Figure 1 **call** ℓ_b asks to open a new session with type:

$$\mathtt{req} \to \mathtt{issuer} : \mathit{req2}(\mathtt{str}).\ \mathtt{issuer} \to \mathtt{req} : \{\ \mathit{yes2}(\mathtt{int}).\mathbf{call}\ \ell_c, \mathit{no2}().\mathtt{end}\ \}$$

3 Lightening global types.

This section describes a function for removing redundant interactions (Definition 1) from (possibly light) global types. It uses lightening, since it adds **call** constructors and declarations. In the next section we will show that the initial and final protocols describe 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 does not appear under a recursion. More precisely, by defining light global contexts (without recursion) as follows:

$$C \quad ::= \quad [\ ] \mid \mathtt{r_1} \to \mathtt{r_2} : \{l_j(S_j).G_j, l(S).C\}_{j \in J}$$

we get:

Definition 1 (Redundant interaction). The interaction $\mathtt{r_1} \to \mathtt{r_2} : l()$ is *redundant* in

$$C[\mathtt{r_1} \to \mathtt{r_2} : l().L\,].$$

Interactions sending only one label and with messages of type unit are needed inside recursions when they terminate the cycle. For example the interaction $\mathtt{r_2} \to \mathtt{r_3} : \mathit{stop}()$ cannot be erased in the type

$$\mu t.\mathtt{r_1} \to \mathtt{r_2} : \{\mathit{goon}(\mathtt{int}).\mathtt{r_2} \to \mathtt{r_3} : \mathit{goon}(\mathtt{int}).t, \mathit{stop}().\mathtt{r_2} \to \mathtt{r_3} : \mathit{stop}().\mathtt{end}\}$$

We define $\mathbb{L}(L)$ as a function for removing redundant interactions in L by decomposing L into separated light global types. The basic idea is that, in order to erase redundant communications, the roles which are the receivers of these communications must get the communications belonging to other branches in separated types. Therefore the result of $\mathbb{L}(L)$ is a new light global type and a set of declarations with fresh names.

The function $\mathbb{L}(L)$ uses the auxiliary function $L \Downarrow \mathtt{r}$ which searches inside the branches of L the first communications with receiver $\mathtt{r}$ and replaces these communications (and all the following types) by **call**s to newly created names, which are associated by declarations to the corresponding types. Therefore also the result of $L \Downarrow \mathtt{r}$ is a new light global type and a set of declarations with fresh names.

Definition 2 (The function $\Downarrow$). The function $L \Downarrow r$ is defined by induction on L:

$$\text{end} \Downarrow r = (\text{end}, \emptyset) \qquad t \Downarrow r = (t, \emptyset) \qquad \textbf{call } \ell \Downarrow r = (\textbf{call } \ell, \emptyset) \qquad \mu t.L \Downarrow r = (\mu t.L, \emptyset)$$

$$(r_1 \to r_2 : \{l_j(S_j).L_j\}_{j \in J}) \Downarrow r = \begin{cases} (L_j, \emptyset) & \text{if } r_2 = r \text{ and} \\ & S_j = \text{unit and} \\ & J = \{j\} \\ (\textbf{call } \ell, \ell = r_1 \to r_2 : \{l_j(S_j).L_j\}_{j \in J}) & \text{if } r_2 = r \text{ and} \\ \quad \text{where } \ell \text{ is a fresh name} & S_j \neq \text{unit or} \\ & J \neq \{j\} \\ (r_1 \to r_2 : \{l_j(S_j).L'_j\}_{j \in J}, \bigcup_{j \in J} D_j) & \text{otherwise} \\ \quad \text{where } L_j \Downarrow r = (L'_j, D_j) \text{ for } j \in J. \end{cases}$$

We remark that the first case of the definition of $\Downarrow$ for a branching type allows this function to eliminate a redundant interaction. Therefore one application of function $\mathbb{L}$ can get rid of more than one redundant interaction, as exemplified below.

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

Definition 3 (The function $\mathbb{L}$). The application of the function $\mathbb{L}$ to $C[r_1 \to r_2 : l().L]$ for eliminating the interaction $r_1 \to r_2 : l().L$ is defined by induction on C:

$$\mathbb{L}(r_1 \to r_2 : l().L) = (L, \emptyset)$$

$$\mathbb{L}(r'_1 \to r'_2 : \{l_j(S_j).L_j, l'(S).C[r_1 \to r_2 : l().L]\}_{j \in J}) =$$

$$\begin{cases} (r'_1 \to r'_2 : \{l_j(S_j).L_j, l'(S).C[L]\}_{j \in J}, \emptyset) & \text{if } r_2 = r'_1 \text{ or } r_2 = r'_2, \\ (r'_1 \to r'_2 : \{l_j(S_j).L'_j, l'(S).L'\}_{j \in J}, \bigcup_{j \in J} D_j \cup D) & \text{if } r_2 \neq r'_1 \text{ and } r_2 \neq r'_2, \\ \quad \text{where } L_j \Downarrow r_2 = (L'_j, D_j) \text{ for } j \in J \\ \quad \text{and } \mathbb{L}(C[r_1 \to r_2 : l().L]) = (L', D) \end{cases}$$

The branching case of the above definitions needs some comments. If the receiver r_2 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 r_2 must receive a communication in all branches l_j and in this case these communications need to be replaced by **call**s to fresh names of light global types. For this reason we compute $L_j \Downarrow r_2$ for all $j \in J$. Moreover we recursively call the mapping $\mathbb{L}$ on $C[r_1 \to r_2 : l().L]$.

We now show two applications of the lightening function, first to the global type G of Figure 1 and then to the light global type obtained as a result of the first application. The application of $\mathbb{L}$ for eliminating the interaction $\text{req} \to \text{store} : no4()$ gives as a result $(L'_a, \ell_c = L_c)$, where L_c is the type associated to ℓ_c in Figure 1 and:

$$\begin{aligned} L'_a = \ &\text{req} \to \text{map} : req1(\text{str}).\text{map} \to \text{req} : \\ &\{ yes1(\text{str}).\text{req} \to \text{issuer} : req2(\text{str}).\text{issuer} \to \text{req} : \\ &\quad \{ yes2(\text{int}).\textbf{call } \ell_c, no2().\text{end}\}, \\ &\ no1().\text{req} \to \text{issuer} : no5().\text{end}\} \end{aligned}$$

Notice that also the redundant interaction $\texttt{req} \to \texttt{store} : no6()$ is erased in L'_a. The same result can be obtained by applying $\mathbb{L}$ for eliminating the interaction $\texttt{req} \to \texttt{store} : no6()$. Now if we apply $\mathbb{L}$ to L'_a for eliminating $\texttt{req} \to \texttt{issuer} : no5()$ we get $(L_a, \ell_b = L_b)$, where L_a and L_b are the types associated to ℓ_a and ℓ_b in Figure 1. So we get all declarations shown in Figure 1.

4 Safety of the lightening function.   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 interactions which can occur in multi-party sessions. The only difference is that recursive types do not reduce, the reasons being that our lightening function does not modify them. 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 $\checkmark$ session termination.

Definition 4 (LTS)**.**

$$\begin{array}{cccc}
[\textsc{call}] & [\textsc{rec}] & [\textsc{red}] & [\textsc{end}] \\[4pt]
\dfrac{\ell = L \in D}{\textbf{call}\ \ell \xrightarrow{\tau}_D L} & \mu t.L \xrightarrow{\checkmark}_D & \texttt{r}_1 \to \texttt{r}_2 : l().L \xrightarrow{\tau}_D L & \texttt{end} \xrightarrow{\checkmark}_D
\end{array}$$

$$[\textsc{act}]$$
$$\dfrac{J \neq \{j\} \text{ or } S_j \neq \mathsf{unit}}{\texttt{r}_1 \to \texttt{r}_2 : \{l_j(S_j).L_j\}_{j \in J} \xrightarrow{\texttt{r}_1 \to \texttt{r}_2 : l_j(S_j)}_D L_j}$$

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

$$\begin{array}{lll}
L \xLongrightarrow{\lambda}_D L' & \text{if} & L \xrightarrow{\tau}{}^*_D L_1 \xrightarrow{\lambda}_D L_2 \xrightarrow{\tau}{}^*_D L' \text{ for some } L_1, L_2 \\[4pt]
L \xLongrightarrow{\lambda \cdot \sigma}_D L' & \text{if} & L \xLongrightarrow{\lambda}_D L_1 \xLongrightarrow{\sigma}_D L' \text{ for some } L_1 \\[4pt]
L \xLongrightarrow{\sigma \cdot \checkmark}_D & \text{if} & L \xLongrightarrow{\sigma}_D L_1 \xLongrightarrow{\checkmark}_D \text{ for some } L_1
\end{array}$$

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

Definition 5 (Semantics)**.** $[\![L]\!]_D = \{\sigma \mid L \xLongrightarrow{\sigma}_D L' \text{ for some } L' \text{ or } L \xLongrightarrow{\sigma}_D\}$.

Soundness of lightening then amounts to show that the function $\mathbb{L}$ preserves the meaning of light global types with respect to the relative sets of declarations. A first lemma shows soundness of the function $\Downarrow$.

Lemma 1. Let L be a light global type with declaration D. If $L \Downarrow \texttt{r} = (L', D')$, then $[\![L]\!]_D = [\![L']\!]_{D' \cup D}$.

Proof. The proof is by induction on L and by cases on Definition 2. The only interesting case is $L = \texttt{r}_1 \to \texttt{r}_2 : \{l_j(S_j).L_j\}_{j \in J}$.

1. If $L \Downarrow \texttt{r} = (L_j, \emptyset)$, then $S_j = \mathsf{unit}$ and $J = \{j\}$. By rule [\textsc{red}] we have $L \xrightarrow{\tau}_D L_j$, thus $[\![L]\!]_D = [\![L_j]\!]_D = [\![L']\!]_D$.

2. If $L \Downarrow \texttt{r} = (\textbf{call}\ \ell, \ell = \texttt{r}_1 \to \texttt{r}_2 : \{l_j(S_j).L_j\}_{j \in J})$, then by rule [\textsc{call}] we have $L' \xrightarrow{\tau}_{\{\ell = L\} \cup D} L$. Thus $[\![L]\!]_D = [\![L']\!]_{\{\ell = L\} \cup D}$.

3. If $L \Downarrow r = (r_1 \to r_2 : \{l_j(S_j).L'_j \Downarrow r\}_{j \in J}, \bigcup_{j \in J} D_j)$, then $L_j \Downarrow r = (L'_j, D_j)$ for $j \in J$. By Definition 5 and rule [ACT] $[\![L]\!]_D = \bigcup_{j \in J} \{ r_1 \to r_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D \}$ and $[\![L']\!]_{\bigcup_{j \in J} D_j \cup D} = \bigcup_{j \in J} \{ r_1 \to r_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L'_j]\!]_{\bigcup_{j \in J} D_j \cup D} \}$. Since by induction $[\![L_j]\!]_D = [\![L'_j]\!]_{D_j \cup D}$, we conclude $[\![L]\!]_D = [\![L']\!]_{D' \cup D}$.

$\square$

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

Proof. The proof is by induction on L and by cases on Definition 3.

1. If $L = r_1 \to r_2 : l().L''$, then $\mathbb{L}(L) = (L'', \emptyset)$. We conclude since by Definition 5 and by rule [RED] $[\![L]\!]_D = [\![L'']\!]_D$.

2. Let $L = C_0[r_1 \to r_2 : l().L'']$ where $C_0 = r'_1 \to r'_2 : \{l_j(S_j).L_j, l'(S).C\}]\}_{j \in J}$. By Definition 5 and by rule [ACT]

$$[\![L]\!]_D \quad = \quad \bigcup_{j \in J} \{ r'_1 \to r'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D \} \cup$$
$$\{ r'_1 \to r'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![C[r_1 \to r_2 : l().L'']]\!]_D \}$$

   (a) If $r_2 = r'_1$ or $r_2 = r'_2$, then

$$[\![L']\!]_D \quad = \quad \bigcup_{j \in J} \{ r'_1 \to r'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D \} \cup$$
$$\{ r'_1 \to r'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![C[L'']]\!]_D \}$$

   We conclude since rule [RED] $[\![C[r_1 \to r_2 : l().L'']]\!]_D = [\![C[L'']]\!]_D$.

   (b) If $r_2 \neq r'_1$ and $r_2 \neq r'_2$, let us assume $L_j \Downarrow r_2 = (L'_j, D_j)$ for $j \in J$ and $\mathbb{L}(C[r_1 \to r_2 : l().L]) = (L'_0, D_0)$. Then

$$[\![L']\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D} \quad = \quad \bigcup_{j \in J} \{ r'_1 \to r'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L'_j]\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D} \} \cup$$
$$\{ r'_1 \to r'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![L'_0]\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D} \}$$

   We conclude since by Lemma 1 $[\![L_j]\!]_D = [\![L'_j]\!]_{D_j \cup D}$ and by induction

$$[\![C[r_1 \to r_2 : l().L'']]\!]_D = [\![L'_0]\!]_{D_0 \cup D}.$$

In the running example we get $[\![G]\!]_\emptyset = [\![L'_a]\!]_{\ell_c = L_c} = [\![L_a]\!]_{\ell_b = L_b \ \ell_c = L_c}$. $\square$

5 Related works and conclusion. In the recent literature global types have been enriched in various directions by making them more expressive through roles [15] or logical assertions [5] or monitoring [4], more safe through security levels for data and participants [8, 7] or reputation systems [6].

The more related papers are [14] and [9]. Demangeon and Honda introduce nesting of 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 [14] are much more expressive than our light types. Carbone and Montesi [9] 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.

In this paper we show how to decompose interactions among multiple participants in order to remove redundant interactions, by preserving the meaning of (light) global types. We plan to implement our lightening function in order to experiment its practical utility in different scenarios.

Acknowledgements We are grateful to Mariangiola Dezani-Ciancaglini for her valuable comments and feedbacks. We also thank PLACES reviewers for careful reading, since we deeply revised this article following their suggestions.

REFERENCES.

[1] Martin Berger & Kohei Honda (2003): *The Two-Phase Commitment Protocol in an Extended π-Calculus*. Available at `http://dx.doi.org/10.1016/S1571-0661(05)82502-2`. EXPRESS'00, 7th International Workshop on Expressiveness in Concurrency.

[2] Jan A. Bergstra, Alban Ponse & Scott A. Smolka: *Handbook of Process Algebra*.

[3] Lorenzo Bettini, Mario Coppo, Loris DAntoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In: *CONCUR*, *LNCS* 5201, Springer, pp. 418–433. Available at `http://dx.doi.org/10.1007/978-3-540-85361-9_33`.

[4] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): *Monitoring Networks through Multiparty Session Types*. In: *FMOODS/FORTE*, *LNCS* 7892, Springer, pp. 50–65. Available at `http://dx.doi.org/10.1007/978-3-642-38592-6_5`.

[5] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-contract for Distributed Multiparty Interactions*. In: *CONCUR*, LNCS, Springer, pp. 162–176. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_12`.

[6] Viviana Bono, Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2012): *A Reputation System for Multirole Sessions*. In: *TGC*, *LNCS* 7173, Springer, pp. 1–24. Available at `http://dx.doi.org/10.1007/978-3-642-30065-3_1`.

[7] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2011): *Information Flow Safety in Multiparty Sessions*. In: *EXPRESS*, *EPTCS* 64, pp. 16–31. Available at `http://dx.doi.org/10.4204/EPTCS.64.2`.

[8] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Tamara Rezk (2010): *Session Types for Access and Information Flow Control*. In: *CONCUR*, *LNCS* 6269, Springer, pp. 237–252. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_17`.

[9] Marco Carbone & Fabrizio Montesi (2012): *Merging Multiparty Protocols in Multiparty Choreographies*. In: *PLACES*, *EPTCS* 109, pp. 21–27. Available at `http://dx.doi.org/10.4204/EPTCS.109.4`.

[10] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Sessions*. *Log. Meth. Comp. Scie.* 8, pp. 1–45. Available at `http://dx.doi.org/10.2168/LMCS-8(1:24)2012`.

[11] Tzu-Chun Chen (2013): *Theories for Session-based Governance for Large-Scale Distributed Systems*. Ph.D. thesis, Queen Mary, University of London, UK. Available at `http://www.di.unito.it/~lambda/biblio/entry-TC2013.html`.

[12] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda & Nobuko Yoshida (2011): *Asynchronous Distributed Monitoring for Multiparty Session Enforcement*. In: *TGC*, *LNCS* 7454, pp. 25–45. Available at `http://dx.doi.org/10.1007/978-3-642-30065-3_2`.

[13] Tzu-Chun Chen & Kohei Honda (2012): *Specifying Stateful Asynchronous Properties for Distributed Programs*. In: *CONCUR*, *LNCS* 7454, pp. 209–224. Available at `http://dx.doi.org/10.1007/978-3-642-32940-1_16`.

[14] Romain Demangeon & Kohei Honda (2012): *Nested Protocols in Session Types*. In: *CONCUR*, *LNCS* 7454, Springer, pp. 272–286. Available at `http://dx.doi.org/10.1007/978-3-642-32940-1_20`.

[15] Pierre-Malo Deniélou & Nobuko Yoshida (2011): *Dynamic Multirole Session Types*. In: *POPL*, ACM, pp. 435–446. Available at `http://doi.acm.org/10.1145/1926385.1926435`.

[16] Yliès Falcone (2010): *You Should Better Enforce Than Verify*. In: *Runtime Verification*, LNCS, Springer, pp. 89–105. Available at `http://dx.doi.org/10.1007/978-3-642-16612-9_9`.

[17] Jifeng He, M. B. Josephs & C. A. R. Hoare (1990): *A Theory of Synchrony and Asynchrony*. Available at `http://www.cs.ox.ac.uk/publications/publication8131-abstract.html`. LaTeXed manuscript.

[18] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denilou & Nobuko Yoshida (2012): *Structuring Communication with Session Types*. In: *To appear in COB 2012*. Available at `http://mrg.doc.ic.ac.uk/publications/structuring-communication-with-session-types/main.pdf`.

[19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen & Nobuko Yoshida (2011): *Scribbling Interactions with a Formal Foundation*. In: *ICDCIT*, *LNCS* 6536, pp. 55–75. Available at `http://dx.doi.org/10.1007/978-3-642-19056-8_4`.

[20] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *ECOOP'91*, *LNCS* 512, pp. 133–147. Available at `http://dx.doi.org/10.1007/BFb0057019`.

[21] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *POPL*, ACM, pp. 273–284. Available at `http://doi.acm.org/10.1145/1328438.1328472`.

[22] Dimitrios Kouzapas, Nobuko Yoshida & Kohei Honda (2011): *On Asynchronous Session Semantics*. In: *FMOODS/FORTE*, pp. 228–243. Available at `http://dx.doi.org/10.1007/978-3-642-21461-5_15`.

[23] *Ocean Observatories Initiative*. `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.

[24] Olaf Owe, Martin Steffen & Arild B. Torjusen (2010): *Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting*. *ENCS* 264(3), pp. 69–84. Available at `http://dx.doi.org/10.1016/j.entcs.2010.12.015`.

[25] *Scribble Project homepage*. `www.scribble.org`.

[26] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2014): *The Scribble Protocol Language*. In: *TGC 2013*. Available at `http://dx.doi.org/10.1007/978-3-319-05119-2_3`.