Asynchronous Global Types in Co-logic Programming

Riccardo Bianchini, Francesco Dagnino

Asynchronous global types in co-logic programming

Riccardo Bianchini and Francesco Dagnino

DIBRIS, Università di Genova

Abstract. Global types are at the core of communication based programming. They allow a high level specification of protocols involving many participants and enforce good safety and liveness properties, such as absence of deadlock, locked participants and orphan messages. In this paper, we describe an implementation of a novel formalism of global types for sessions with asynchronous communications in co-logic programming, where we use coinduction to properly handle the coinductive syntax of global types and processes. We also define a simple query language to write sessions and global types, providing primitives for type checking.

Keywords: Global types · Prolog · Coinduction.

1 Introduction

We describe an implementation in *co-logic programming* [9,11,1] of a novel formulation of global types for asynchronous sessions, described in a companion paper [4]. Co-logic programming is an extension of logic programming where predicates can be marked as coinductive. In this case, resolution relies on a mechanism of cycle detection which gives (successful) termination when the same goal is encountered twice.

The benefits of this work are twofold: on one hand, to provide an implementation of the type system described in [4], and a simple user interface for making related queries. On the other hand, global types and related judgments provide a very interesting and challenging case study for co-logic programming, since their encoding forces to clearly understand and express the either inductive or coinductive nature of definitions, and the related termination issues. Notably, sometimes inductive predicates are adequate, sometimes they need to be implemented, rather than directly, as the negation of a predicate defined coinductively, in other cases it is necessary to use the coinductive extension of SWI-Prolog. Finally, in some cases, a by-hand cycle detection mechanism is needed, since neither a standard inductive definition, nor a coinductive definition using built-in cycle detection are enough to ensure termination. These issues are discussed in detail in Sect. 5. We used SWI-Prolog [12] version 8.2.2 for x64-win64.

The tool is composed of two components:

– the core part, that is, the Prolog implementation of definitions in [4], such as sessions, global types, projections, and typing judgments

– the query language, which provides a more user-friendly syntax, and a simple typechecking phase to avoid inconsistencies in the user's code.

Sect. 2 reports the definitions from [4] implemented in the tool. Sect. 3 is a brief presentation of co-logic programming. Sect. 4 describes the query language, and Sect. 5 the Prolog implementation, discussing termination issues. Finally, in Sect. 6 we summarize the contribution and discuss future developments. The complete code, and instructions for using the prototype, can be found at `https://github.com/RiccardoBianc/Asynchronous-global-types-implementation`.

2 Global types for asynchronous sessions

We briefly summarize the formulation of global types for asynchronous sessions introduced in [3], and subsequently extended in [4], reporting the formal definitions implemented in the tool. The key idea in [3] is to directly handle asynchrony at the level of global types, in the sense that an output and the corresponding input operation are modeled by distinct type constructors. In this way, we can directly assign a global type to an asynchronous session without the need of asynchronous subtyping, but this comes at the cost that not all global types ensure the desired properties, hence a notion of *well-formedness* becomes crucial in our setting.

Another novelty with respect to classical presentations [5,2] of global types is that a *coinductive approach* is adopted. Namely, processes and types with an infinite behaviour are expressed as infinite regular terms, rather than by an explicit fixed-point operator, and, correspondingly, functions handling them, e.g., the projection, are also defined coinductively. This feature makes the implementation in co-logic programming very natural, as shown in the following.

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

$$\mathsf{P} ::=_\rho \mathsf{p}!\{\lambda_i.\mathsf{P}_i\}_{i\in I} \mid \mathsf{p}?\{\lambda_i.\mathsf{P}_i\}_{i\in I} \mid \mathbf{0} \qquad I \neq \emptyset, \lambda_j \neq \lambda_h \text{ for } j \neq h$$

The symbol $::=_\rho$ indicates that the productions should be interpreted *coinductively*, rather than inductively as in the standard case. That is, they define possibly infinite terms. However, we assume such infinite terms to be *regular*, that is, with finitely many distinct sub-terms.

A process of shape $\mathsf{p}!\{\lambda_i.\mathsf{P}_i\}_{i\in I}$ (*internal choice*) sends to p one of the labels in a set, and then behaves differently depending on the sent label. A process of shape $\mathsf{p}?\{\lambda_i.\mathsf{P}_i\}_{i\in I}$ (*external choice*) waits for receiving from p one of the labels in a set, and then behaves differently depending on the received label. An internal choice which is a singleton is simply written $\mathsf{p}!\lambda.\mathsf{P}$, and $\mathsf{p}!\lambda.\mathbf{0}$ is abbreviated $\mathsf{p}!\lambda$, and analogously for an external choice.

Queues are sequences of *messages*, which are triples consisting of a sender, a label, and a receiver, as shown below:

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

(Multiparty) sessions consist of pairs participant/process composed in parallel, each with a different participant, and a queue. That is, a session has shape $\mathbb{N} \parallel \mathcal{M}$, where

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

For example consider the session:

$$\mathsf{p}[\![\, \mathsf{q}!\lambda.\mathsf{q}?\lambda' \,]\!] \parallel \mathsf{q}[\![\, \mathsf{p}!\lambda'.\mathsf{p}?\lambda \,]\!] \parallel \emptyset \tag{1}$$

where each of the participants p and q wishes to first send a message to and then receive a message from the other one. In a synchronous setting, this session would be stuck, because a communication arises from the synchronisation of an output with a matching input, and here the output $\mathsf{q}!\lambda$ of p cannot synchronise with the input $\mathsf{p}?\lambda$ of q, since the latter is guarded by the output $\mathsf{p}!\lambda'$. Symmetrically, the output $\mathsf{p}!\lambda'$ of q cannot synchronise with the input $\mathsf{q}?\lambda'$ of p. Instead, in an asynchronous setting, p could put its message for q on the queue and q could read it after putting its message for p on the queue and viceversa.

Type system Classical global types describe the interaction in a session with communications specifying the sender of the message, its receiver, and the sent message. So a communication embeds both the sending and the receiving of the message. Hence, it is not possible to assign a global type, e.g., to the session above, since this type should specify that either the communication in which p sends λ to q or the one in which q sends λ' to p takes place first. In the global types of [3], instead, communications are split into outputs and inputs, as follows:

$$\mathsf{G} ::=_\rho \mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I} \mid \mathsf{pq}?\lambda.\mathsf{G} \mid \mathsf{End} \qquad \mathsf{p} \neq \mathsf{q}, I \neq \emptyset, \lambda_j \neq \lambda_h \text{ for } j \neq h$$

A global type of shape $\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}$ specifies that p sends to q one of the labels in a set, and then an interaction takes place which depends on the sent label. A global type of shape $\mathsf{pq}?\lambda.\mathsf{G}$ specifies that q receives from p the label λ, and then the interaction described by G takes place. As processes, global types are defined coinductively, so that infinite global types are allowed, but only of regular shape. The *players* of a global type are those p occurring as senders in outputs ($\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}$) or receivers in inputs ($\mathsf{qp}?\lambda.\mathsf{G}$).

With these types, it is possible to describe the asynchronous session (1) with the type

$$\mathsf{pq}!\lambda.\mathsf{qp}!\lambda'.\mathsf{pq}?\lambda.\mathsf{qp}?\lambda'.\mathsf{End}$$

or with the others obtained by swapping the order of outputs or inputs, that is,

$$\mathsf{qp}!\lambda'.\mathsf{pq}!\lambda.\mathsf{pq}?\lambda.\mathsf{qp}?\lambda'.\mathsf{End}$$
$$\mathsf{pq}!\lambda.\mathsf{qp}!\lambda'.\mathsf{qp}?\lambda'.\mathsf{pq}?\lambda.\mathsf{End}$$
$$\mathsf{qp}!\lambda'.\mathsf{pq}!\lambda'.\mathsf{qp}?\lambda'.\mathsf{pq}?\lambda.\mathsf{End}$$

Configuration types are pairs $\mathsf{G}\|\mathcal{M}$ where G is a global type and $\mathcal{M}$ is a queue. A configuration type describes a session in which some participant sent a message that is not yet read from its receiver. *Well-formedness* of $\mathsf{G}\|\mathcal{M}$ is defined as the conjunction of the following properties:

- $\mathsf{G}\|\mathcal{M}$ is *input/output matching*, that is, every message put on the queue will be eventually read and every enabled input should find a corresponding message in the queue.
- G is *bounded*, that is, the first occurrence as player of a participant, if any, is at a bounded depth in all paths, ensuring that no player remains stuck forever.
- The projection of G on each player is well-defined (explained below).

The tool implements these properties as described in [4], which significantly enlarge the class of typable sessions with respect to [3].

The notion of projection is a key one in type systems for multiparty sessions. Usually [5,6], global types are projected onto *local types* and local types are assigned to processes. In the simple calculus in [4], global types can be directly projected onto processes, as in [8,3]. Projection computes, starting from a global type G, the (most general) process P associated with a single participant p. This is modeled by the judgment $\mathsf{G}\upharpoonright\mathsf{p} \mapsto \mathsf{P}$, which is defined coinductively.

Example 1. For instance, the previously considered global type

$$\mathsf{G} = \mathsf{pq}!\lambda.\mathsf{qp}!\lambda'.\mathsf{pq}?\lambda.\mathsf{qp}?\lambda'.\mathsf{End}$$

is projected to processes $\mathsf{P} = \mathsf{q}!\lambda.\mathsf{q}?\lambda'.\mathbf{0}$ for participant p and $\mathsf{Q} = \mathsf{p}!\lambda'.\mathsf{p}?\lambda.\mathbf{0}$ for participant q.

As already mentioned, the notion of well-formedness is crucial in this setting as not all global types ensure desirable properties of asynchronous sessions. We show below some examples of such global types explaining why they are not well-formed.

Example 2. The following global type describes a deadlocked session, as q is blocked waiting for the message λ_2:

$$\mathsf{G} = \mathsf{pq}!\lambda_1.\mathsf{qp}!\lambda_2.\mathsf{pq}?\lambda_2.\mathsf{qp}?\lambda_2.\mathsf{G}$$

This type is not input/output matching, as the input $\mathsf{pq}?\lambda_2$ does not match any previous output, hence G is not well-formed.

Example 3. The following global type describes a session where the participant r can wait forever, because p and q can exchange the message λ_1 forever:

$$\mathsf{G} = \mathsf{p}\,\mathsf{q}!\{\lambda_1.\mathsf{pq}?\lambda_1.\mathsf{G},\ \lambda_2.\mathsf{pq}?\lambda_2.\mathsf{qr}!\lambda.\mathsf{qr}?\lambda.\mathsf{G}\}$$

This type is not well-formed as it is not bounded precisely because of the infinite path $\mathsf{pq}!\lambda_1.\mathsf{pq}?\lambda_1.\mathsf{pq}!\lambda_1\ldots$, which does not involve r.

Example 4. The following global type describes a session where the first message is never read:

$$\mathsf{G} = \mathsf{qp}!\lambda_1.\mathsf{G}_1 \qquad \mathsf{G}_1 = \mathsf{pq}!\lambda.\mathsf{pq}?\lambda.\mathsf{G}_1$$

This type is not input/output matching, as the output $\mathsf{qp}!\lambda_1$ is not matched by any subsequent input, hence it is not well-formed.

Boundedness and projection, together with their Prolog implementation, will be described in more detail in Sect. 5.

The typing judgment $\mathbb{N}\|\mathcal{M} : \mathsf{G}\|\mathcal{M}$ checks that the session $\mathbb{N}\|\mathcal{M}$ is consistent with the global protocol represented by the configuration type $\mathsf{G}\|\mathcal{M}$. The judgment is derived from the following conditions, for $\mathbb{N} = \mathsf{p}_1[\![\,\mathsf{P}_1\,]\!] \parallel \cdots \parallel \mathsf{p}_n[\![\,\mathsf{P}_n\,]\!]$:

- For each participant p_i, the associated P_i should be *consistent* with that obtained as projection of the global type. That is, the protocol specified through the global type can be more general than the process in the session, as formalized by a preorder on processes.
- The players of the global type are a subset of the participants $\mathsf{p}_1, \ldots, \mathsf{p}_n$ of the session. The converse is not required, so that, if a session is well-typed, then the session obtained adding participants with inactive processes is well-typed as well.

Whereas the tool is devoted to the implementation of the type system, hence of the syntactic definitions and judgments described so far, [4] also provides an *asynchronous operational semantics* for multiparty sessions, by means of a labelled transition system, and proves that the type system ensures the following properties of computations: deadlock-freedom (in every reachable state of computation, the session is either terminated or it can move); input lock-freedom (every component wishing to do an input will eventually do so); orphan-message-freedom (every message stored in the queue is eventually read).

3 Co-logic programming

A limit of standard logic programming is that we cannot define predicates on non-well-founded structures, such as infinite lists. To overcome this, logic programming has been extended to support coinduction by *coinductive logic programming* [9,11,1], where terms are coinductively defined, that is, can be infinite, and predicates are coinductively defined as well. Possibly infinite terms are represented by finite sets of equations between finite terms. For instance, the equation `L = [1,2|L]` represents the infinite list `[1,2,1,2,...]`. On the other hand, the infinite list of odd numbers *cannot* be represented by a finite set of equations.

Moreover, standard SLD resolution is replaced by co-SLD resolution [11,1], which, roughly speaking, keeps trace of already encountered goals, called *coinductive hypotheses*, so that, when a goal is found the second time, it is considered successful.

A drawback of coinductive logic programming is that *all* predicates are interpreted coinductively, whereas in applications it is often the case that predicates

to be interpreted inductively and coinductively should coexist. To overcome this issue, *co-logic programming* [10] marks predicates as either inductive or coinductive; however, no mutual recursion is allowed between an inductive and a coinductive predicate, that is, stratification is needed. Hence each layer can be interpreted as the least or greatest fixed point, respectively, of an inference system where the lower levels are assumed as axioms. This approach of marking predicates is supported by SWI-Prolog, the Prolog environment used for the implementation.

4 Query language

Together with the implementation, we provide a high-level query language which can be used to easily check the judgments described in Sect. 2. A program in this language consists of *groups*, each one consisting of many *tests*. Both groups and tests have names. For instance, the program below consists in a single group, composed of two tests. In the first test, the global type and the processes are those of Example 1, hence all queries succeed. In the second test, the global type is that of Example 3, and the query `not bounded G` succeeds, indeed the global type is not bounded.

```
Test_Group[
Example_1{
Process P = q!L; q?L1; 0
Process Q = p!L1; p?L; 0
GlobalType G = p>q!L; q>p!L1; p>q?L; q>p?L1; End
Session S = p[P] | q[Q] | Empty
io-match G|Empty
bounded G
proj(G,q) ==  Q
wf G|Empty
S has type G|Empty
}

Example_3{
GlobalType G = p>q!
     {
     L1; p>q?L1; G,
     L2; p>q?L2; q>r!L; q>r?L; G
     }
not bounded G
}

]
```

Each test consists of a list of declarations, followed by a list of queries.

Declarations begin with a keyword for the kind of declared entity: `Process` for processes, `GlobalType` for global types, `Queue` for queues, and `Session` for sessions. The syntax for such entities closely follows that of Sect. 2, apart that we use the

separator `>` to suggest the direction of the communication. The empty queue is represented by the constant `Empty`. The declarations can be mutually recursive. The tool performs a rudimentary typechecking, e.g., rejecting a program where a declared process is used as a queue.

Queries correspond to judgments described in Sect. 2. In particular:

- `io-match G|M` checks that the configuration type `G|M` is input/output matching
- `bounded G` checks that the global type `G` is bounded
- `proj(G,p) == P` checks that the projection of the global type `G` on `p` is `P`
- `exists-proj(G,p)` checks that the projection of the global type `G` on `p` is well-defined
- `exist-all-proj G` checks that all the projections of the global type `G` are well-defined
- `wf G|M` checks that the configuration type `G|M` is well-formed.
- `S has type G|M` checks that `S` is well-typed with respect to the configuration type `G|M`
- for each query, it is also possible to check that its negation holds by prepending `not`.

Again, the tool checks that entities are used in the queries accordingly to their declaration. For instance, in the typing query it is checked that the first argument is a session and the second argument is a configuration type.

The query language is parsed using ANTLR [7], generating a parsing tree which is used to obtain the Prolog code. Then, the tool executes the Prolog file directly using the Java `Runtime` standard library and its method `exec` to execute string commands in separate processes and, finally, the results are shown.

5 Prolog implementation

We illustrate some fragments of Prolog code, chosen simple for space limits, yet providing the flavour of the kind of issues to be faced in the implementation. First of all we mention that global types, being coinductively defined, are implemented by (equations between) terms `G` of shape either `output_type(A,B,[L-G|LGs]))`, or `input_type(A,B,L,G))`, or `end`, where `A`, `B` are participants, `L` are labels, and `L-G` and `LGs` are pairs ⟨label,global type⟩ and lists of such pairs, respectively.

The first two are examples of cases where, to implement a coinductively defined judgment, it is not adequate to just use a coinductive predicate, but other strategies need to be used.

Consider the definition of `player(G,A)`, checking whether the participant `A` occurs as player in `G`. On an infinite (regular) global type, an inductive definition of `player` would not terminate in the negative case, while a coinductive definition would be not correct, since it would be successful, when finding a cycle (that is, the same global type), for an arbitrary argument. The solution is to define `player` as the negation of a predicate `not_player`.

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

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

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

not_player(end,_).
```

The predicate visits the global type and checks that at each node the participant argument is not a player. This predicate being coinductive, when a cycle is found the call succeeds, hence the predicate `player` fails, correctly, in the negative case. On the other hand, in the positive case (an argument which is a player) the predicate `not_player` finitely fails, hence the predicate `player` succeeds.

In other cases, a by-hand cycle detection mechanism is needed. An example is the definition of `players(G,As)`, computing the set of players `As` of `G`. On an infinite (regular) global type, again an inductive definition would not terminate. On the other hand, a coinductive definition would accept *all the supersets* of the players, implementing a slightly different concept. The solution is to define `players` using an additional parameter, the list `Gs` of already encountered global types, initially empty.

```
players(Gs,G,[]) :-
    member(G,Gs).

players(Gs,output_type(A,B,LGs),As) :-
    \+member(output_type(A,B,LGs),Gs),
    players_list([output_type(A,B,LGs)|Gs],LGs,Bs),
    union(A,Bs,As).

players(Gs,input_type(A,B,L,G),As) :-
    \+member(input_type(A,B,L,G),Gs),
    players([input_type(A,B,L,G)|Gs],G,Bs),
    union(B,Bs,As).

players(_,end,[]).
```

The predicate visits the global type, and, at each node, the current global type is added to the list if not present yet, otherwise a cycle is detected and the result is the empty set. At each step, the result is the union of the players of the subterms and of the current node; since, when a cycle is found, the result is only the empty set, the only solution is exactly the set of players, rather than all the supersets as in the coinductive case.

As more significant and involved examples, we describe the implementation of the boundedness check and of the projection judgment. This is interesting since it is not trivial to design a concrete algorithm from these abstract definitions.

Boundedness Global types can be naturally seen as trees. We use ξ to denote a *path* in global type trees, that is, a possibly infinite sequence of communications $\mathsf{p}\,\mathsf{q}!\lambda$ or $\mathsf{p}\,\mathsf{q}?\lambda$. With ξ_n we represent the n-th communication in the path ξ, where $0 \leq n < x$ and $x \in \mathbf{N} \cup \{\omega\}$ is the length of ξ. With ϵ we denote the empty sequence and with $\cdot$ the concatenation of a finite sequence with a possibly infinite sequence. The function Paths gives the set of *paths* of global types, which are the greatest sets such that:

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

The definition of boundedness is based on the concept of *depth* of a player. Let G be a global type. For $\xi \in \mathsf{Paths}(\mathsf{G})$, set $\mathsf{depth}(\xi, \mathsf{p}) = \inf\{n \mid \mathsf{play}(\xi_n) = \mathsf{p}\}$, and define $\mathsf{depth}(\mathsf{G}, \mathsf{p})$, the *depth* of p in G, as follows:

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

Note that, if p is a player of G, but it does not occur as player in some path ξ of G (that is, $\mathsf{p} \neq \mathsf{play}(\xi_n)$ for all $n \in \mathbf{N}$), then $\mathsf{depth}(\xi, \mathsf{p}) = \inf \emptyset = \infty$, modelling the fact that p may wait forever.

A global type G is *bounded* if $\mathsf{depth}(\mathsf{G}', \mathsf{p})$ is finite for all $\mathsf{p} \in \mathsf{players}(\mathsf{G})$ and all types G' which occur in G.

This check is implemented by the predicate `bounded` below.

```
bounded(G) :-
    players(G,As),
    bounded_list(G,As).

bounded_list(G,[A]) :-
         all_finite_depth(G,A).

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

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

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

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

```
          not_player(G,A).

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

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

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

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

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

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

Note the difference between `player(G,A)`, holding if `A` occurs as player in some path, and `finite_depth(G,A,[])`, holding if `A` occurs as player in each path (or is not a player at all). The negation of the former is a universal property (`A` never occurs as a player), which can be defined by a coinductive predicate, namely, `not_player(G,A)`, so to rely on the built-in cycle detection mechanism offered by SWI-Prolog. The negation of the latter is an existential property (`A` is a player, and it does not occur in some path) which cannot be defined by a coinductive predicate. The solution is to use a by-hand cycle detection mechanism as described above.

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

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

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

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

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

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

$$[\textsc{Out-Rcv}]\ \frac{\mathsf{G}_i \upharpoonright \mathsf{q} \mapsto \mathcal{C}[\mathsf{p}?\lambda_i.\mathsf{P}_{i,j}]_{j\in J} \quad \forall i \in I}{(\mathsf{pq}!\{\lambda_i.\mathsf{G}_i\}_{i\in I}) \upharpoonright \mathsf{q} \mapsto \mathcal{C}[\mathsf{p}?\{\lambda_i.\mathsf{P}_{i,j}\}_{i\in I}]_{j\in J}}\ \mathsf{q} \in \mathsf{play}(\mathsf{G}_i)\ i \in I$$

Fig. 1. Projection: contexts and example of rule

```
all_finite_depth(end,_).
```

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

Projection The definition of projection $\mathsf{G} \upharpoonright \mathsf{q} \mapsto \mathsf{P}$ uses process contexts with an arbitrary number of holes indexed with natural numbers, where each hole has a different index. Given a context $\mathcal{C}$ with holes indexed in J, we denote by $\mathcal{C}[\mathsf{P}_j]_{j\in J}$ the process obtained by filling the hole indexed by j in P_j, for all $j \in J$. In Fig. 1 we report the definition of contexts and one rule, namely, the one defining the projection of an output choice on the receiver q, in the case it is a player in the global type (the projection on a non-player is always the inactive process).

The rule states that the projection is well-defined if the projections on q of the branches G_i, for $i \in I$, give processes which can be consistently combined to provide the resulting projection. More precisely, they have a common structure, modelled by a multi-hole context $\mathcal{C}$, where, for each hole $j \in J$, this is filled by a different process subterm in the projection of each branch. The process filling the j-th hole of the projection of branch i must start with an input from p and label λ_i. In this way, the processes in the j-th holes of all branches can be combined in an external choice, which is used to fill the context in the resulting projection.

Note that the rule just assumes the *existence* of context $\mathcal{C}$, whereas in the implementation this context, if any, should be *constructed*. We show below how this is achieved, in a simplified version, yet illustrating all the key features of real code:

```
projection(output_type(A,B,[L1-G1,L2-G2|LGs]),B,P) :-
        player(output_type(A,B,[L1-G1,L2-G2|LGs]),B),
        projection_list([L1-G1,L2-G2|LGs],B,[P1,P2|Ps]),
        build_context(P1,P2,A,L1,L2,Context),
        pairs_keys(LGs,Ls),
        check_each_process(Context,A,[L1,L2|Ls],
                           [P1,P2|Ps],Fillings),
        build_process_result(Context,Fillings,A,P).
```

This clause models the case when the output choice has two or more branches; the case of only one branch is handled by an ad-hoc clause that does not build any context, because this is not necessary. After checking that the participant `B` is a player in the global type, the predicate `projection_list` computes the

projections on `B` of all the subterms. Then, the predicate `build_context`, declared as coinductive, is used to build the context $\mathcal{C}$. This predicate visits at the same time the processes `P1` and `P2` obtained as projection of the first two branches, checking that they are equal node by node, and reporting this common structure in `Context`, until one among these three conditions occurs:

- Both processes are inactive.
- Both are an input node receiving from `A` the corresponding label (`L1` and `L2`, respectively), as explained above.
- Both are encountered for a second time.

In the first case the resulting context is the inactive process. In the second case the result is a hole context, since holes model the different process subterms after the input. In the third case, the query is successful, as it happens in a coinductive predicate, hence the context result turns out to be cyclic as well. After building the context, the predicate `pairs_keys` obtains the list of the branch labels. Then, the predicate `check_each_process` checks that all the processes computed as projections of the subterms can be obtained by filling the holes in the context, producing `Fillings`, a list with an element for each branch $i \in I$, which is in turn a list with an element $\lambda_i.\mathsf{P}_{i,j}$ for each hole $j \in J$. Finally, the predicate `build_process_result` builds the final process result, filling the context as explained above.

6 Conclusion

The work described in this paper is an implementation in co-logic programming of a novel formulation of global types for asynchronous session, where asynchrony is expressed at the level of the type system. The tool has been developed in parallel with the theoretical investigation, thus it has been very useful to check the proposed definitions, finding in some cases subtle bugs. The main challenge has been to solve termination problems, naturally arising since we had to deal with infinite, yet regular, structures. Our solutions, only illustrated on simple examples in this short paper, are based on coinductive techniques mixed with inductive ones, also employing in some cases user-defined cycle detection mechanisms.

The tool could be improved in many directions. First of all, the current implementation is a prototype, which should be refined and equipped with a suitable user interface to become more usable. Currently, we only allow in the user language queries with a yes/no answer. Queries computing a result, e.g., the projection of a global type on a participant, can be easily performed at the Prolog level, but the tool still lacks a reverse translation to show the user language version of the Prolog answer. Finally, another direction could be a more efficient implementation, for example using a language such as C++.

Acknowledgements We thank all the anonymous referees for their careful reading and useful comments, which helped us improve the paper. We are also grateful

to Paola Giannini and Elena Zucca for their many suggestions to make the presentation clearer.

References

1. Ancona, D., Dovier, A.: A theoretical perspective of coinductive logic programming. Fundamenta Informaticae **140**(3-4), 221–246 (2015). https://doi.org/10.3233/FI-2015-1252
2. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) Concurrency Theory, CONCUR'08. Lecture Notes in Computer Science, vol. 5201, pp. 418–433. Springer (2008). https://doi.org/10.1007/978-3-540-85361-9_33
3. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Global types and event structure semantics for asynchronous multiparty sessions. CoRR **abs/2102.00865** (2021), `https://arxiv.org/abs/2102.00865`
4. Dagnino, F., Giannini, P., Dezani-Ciancaglini, M.: Deconfined global types for asynchronous sessions. In: COORDINATION 2021 - Coordination Models and Languages. Lecture Notes in Computer Science, Springer (2021)
5. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) ACM Symposium on Principles of Programming Languages, POPL'06. pp. 273–284. ACM Press (2008). https://doi.org/10.1145/1328438.1328472
6. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
7. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013)
8. Severi, P., Dezani-Ciancaglini, M.: Observational Equivalence for Multiparty Sessions. Fundamenta Informaticae **170**(1-3), 267–305 (2019). https://doi.org/10.3233/FI-2019-1863
9. Simon, L.: Extending logic programming with coinduction. Ph.D. thesis, University of Texas at Dallas (2006)
10. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) Automata, Languages and Programming, 34th International Colloquium, ICALP'07. Lecture Notes in Computer Science, vol. 4596, pp. 472–483. Springer (2007)
11. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczynski, M. (eds.) Logic Programming, 22nd International Conference, ICLP'06. Lecture Notes in Computer Science, vol. 4079, pp. 330–345. Springer (2006)
12. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)