Softly safely spoken: Role playing for Session Types

Elena Giachino*
Dipartimento di Informatica
Università degli Studi di Torino
Torino, Italy
PPS - Université Paris Diderot
Paris, France
giachino@di.unito.it

Matthew Sackman, Sophia Drossopoulou, Susan Eisenbach,
Department of Computing
Imperial College
London, England
ms@doc.ic.ac.uk, sd@doc.ic.ac.uk, sue@doc.ic.ac.uk

Abstract

Session types have made much progress at permitting programs be statically verified concordant with a specified protocol. However, it is difficult to build abstractions of, or encapsulate Session types, thus limiting their flexibility. Global session types add further constraints to communication, by permitting the order of exchanges amongst many participants to be specified. The cost is that the number of participants is statically fixed.

We introduce Roles in which, similarly to global session types, the number of roles and the conversations involving roles are statically known, but participants can dynamically join and leave roles and the number of participants within a role is not statically known. Statically defined roles which conform to a specified conversation can be dynamically instantiated, participants can be members of multiple roles simultaneously and can participate in multiple conversations concurrently.

1 Introduction

Communication is the single most important faculty of computing beyond the definition of computation itself. Of course, communication is built into computation itself, be it defined by the Turing Machine or the λ-calculus, by the ability to pass names, values or state between instructions. Increasingly, as programs are targeting distributed and highly parallel environments, safe communication between such programs is of the utmost importance and subject to a great deal of study.

Session types [8, 14] have provided a means to define protocols, to parametrise communication channels by such protocols, and to statically enforce that use of those communication channels matches the protocol specified. Thus upon any given dyadic communication channel, the two participants will not both be blocked, each waiting for the other to speak; and a value sent as one type will be received as the same type. Session types cater for branching and looping control-flow structures, and there have been several implementations of session types in a number of different languages [11, 13, 6, 7, 12, 10].

A language that supports session types can allow many channels to be defined and used at the same time, and enforces that communication actions upon those channels obey the requirements of their session types. However, the session types only specify the behaviour of a single channel at a time, and so it is difficult to reason about or understand the communication within a larger system, with many channels in use often reusing the same session type. Global session types [4, 9] solve this problem by allowing the protocol to be specified not in terms of channels, but in terms of participants, the types of values to be sent between participants and the order in which individual communication actions are to occur. Thus it is very easy to see the communication of the whole system. The global session type is then projected to a series of individual session types for the channels that each participant needs to use.

The disadvantage to this is that the global session type now specifies not only the communication, but also the number of participants involved. Without global session types, new participants can be created dynamically, their names can be communicated amongst existing participants, and new channels can be dynamically created to include the new-comers into the existing conversation. We feel that this is a valuable property that caters for a vast number of important use-cases.

*Partially supported by the Fondation Sciences Mathématiques de Paris.

One further limitation of both session types and global session types is that it is not easy to encode broadcast semantics. One-to-many and many-to-one (and many-to-many) are all possible (though only through specifying multiple consecutive send and receive actions), but only when the number of participants on each side is known. In the case of global session types, this must be known statically, but even without global session types, polling a large number of channels is not a well supported feature, and it is hard, if not impossible, to permit new channels to be created and received upon during an on-going multi-receive, or to release channels to participants that have died. Very few implementations of session types even cater for receiving on a set of channels ([11] is one of the few that does).

We introduce a language with support for Roles. Roles can contain an arbitrary number of participants, and participants are free to join and leave Roles at any time, subject to type-safety constraints. Communication is then specified by a `conversation`, which defines names for roles, names for channels between roles, and gives the dyadic session types for those channels. Because channels are between roles, and not between individual participants, communication is by definition broadcast, with many-to-many semantics. This design restores some of the dynamic flexibility of session types and allows us to use Roles naturally in a number of useful examples, but also retains the high-level declaratory communication plan of global session types.

2 The Auction

The design of our language has been motivated by a number of use-cases, one of which is an *ebay*-style auction. There are three distinct sorts of participants (or *roles*) in such an auction: the auctioneer, the bidders and the audience. As is the case with a real *ebay* auction, bidders can dynamically join and leave the auction after the start and before the end. At any point during the auction, a member of the audience may become a bidder by placing a bid, and bidders who are no longer interested in making any further bids may become members of the audience. Our language does not place any constraints on the number of participants within a role; this is an important feature for being able to model this auction without having to statically prescribe the number of bidders.

The auction takes place in a room in which everyone bar the auctioneer is a member of the audience. The auctioneer names the item, and states the reserve price. After this, bidders may call out their bids (implicitly becoming bidders rather than plain members of the audience). The bidders can't see who else is bidding, nor hear each other's bids, but they can hear the auctioneer announce what the current leading bid is and by whom, whenever a new high bid is received. Like *ebay*, the auction is timed so bids can received up until the time at which the auction ends.

$$
\begin{array}{l}
\texttt{conversation}\ \mathit{Auction}\ \{ \\
\quad \texttt{role}\ \mathit{Auctioneer} \\
\quad \texttt{role}\ \mathit{Audience} \\
\quad \texttt{role}\ \mathit{Bidders} \\
\quad \texttt{channel}\ \mathit{k1}\ \mathit{k1'}\ (\mathit{Auctioneer}, \mathit{Audience}) : !\mathit{Item}.\,!\,\mathit{Price}.\mu T.\,!\,(\mathit{Price}, \mathit{Bool}, \mathit{Pid}).T \\
\quad \texttt{channel}\ \mathit{k2}\ \mathit{k2'}\ (\mathit{Auctioneer}, \mathit{Bidders}) : \mu\, T.?\,\mathit{Price}.T \\
\}
\end{array}
$$

First we define the *conversation*. The conversation exists to name a specification of roles and channels. When a conversation is instantiated, all the roles within it are created (initially without participants), as are the channels between the roles. A conversation is itself a type, and similar to a class in an object-oriented sense.

new Auction(a);

```
join (a.Audience);
a.k1′ ? (i : Item) [ ];
μ X.a.k1′ ? (current : Price, done : Bool,
              leader : Pid) [ ];
  μ Y.if done
       then { }
       else {if interested (i, current)
               then {join (a.Bidders);
                     a.k2′ ! (myNextBid (i, current),
                              self ());
                     leave (a.Bidders);
                    }
               else { }
            a.k1′ ? (current : Price, done : Bool,
                     leader : Pid) [ ];
            if done then { }
               else if (leader ≡ self ())
                       then X;
                       else Y;
       }
```

```
join (a.Auctioneer);
currentPrice = reservePrice;
leader = null;
a.k1 ! (Item);
a.k1 ! (currentPrice, False, leader);
μ X.if isFinished ()
       then {a.k1 ! (currentPrice, True,
                     leader); }
       else {a.k2 ? (bid : Price, bidder : Pid)
               [gotBid (bid, bidder)];
               gotBid (bid, bidder);
             X; }
function gotBid (bid, bidder){
  if bid > currentPrice
     then {a.k1 ! (bid, isFinished (),
                   bidder);
           currentPrice = bid;
           leader = bidder; }
     else { }
}
```

Figure 1: Example programs for the bidders (left) and auctioneer (right)

The conversation declares its name (*Auction*), the names of the three roles within each *Auction* conversation (i.e. *Auctioneer*, *Audience* and *Bidders*) and creates two channels. One channel is between the Auctioneer and the Audience and the other channel is between the Auctioneer and the Bidders. The channel between the *Auctioneer* and *Audience* is called *k1* when it is being used by the *Auctioneer* and is called $k1'$ when it is being used by the *Audience*. That is, channel endpoints have distinct names. Also specified with each channel is its session type. The session type is given for the left endpoint (e.g. *k1* and *k2*), and the dual is calculated in the normal way for the other channel end point (e.g. $k1'$ and $k2'$). The session types for channels are as standard, with send, receive and recursion (defined using μ as standard, and thus must be contractive), but we omit branching intentionally as it can be simulated through `join()` and `leave()`, and because it is unclear what the desired behaviour should be when multiple participants within the same role select different branches within the same offer action.

All audience members share the same behaviour: they can each choose whether or not to bid and how much to bid, but the communication actions must all conform to the session types declared with the channels. Members of the audience will receive the item name, and will then repeatedly receive a tuple, containing the new leading bid, a boolean indicating whether the auction has finished yet, and the name of the leading bidder. From the bidders, the auctioneer will repeatedly receive new bids.

Example programs are shown for the bidders and auctioneer in figure 1 in a π-calculus style syntax. The bidder starts by joining the audience. This allows them to hear the auctioneer announcing the item and the reserve price. Whilst the auction is still in progress, each bidder decides for themselves whether they are interested in making a bid on the current item (*i*) given its current price (*current*). If so, they will join the *Bidders* role and send their bid. They then leave the *Bidders* role and wait to see what happens. Eventually, the auctioneer will announce the new leading bid. If the new leading bid is the bid that the bidder just placed then the bidder returns to waiting for the next bid to be made (or exits if the time limit

is up).

All participants within a role must obey the session types of the role's channels. If the session type of a channel indicates the next action is to send a value on that channel then *all* the participants within that role must send upon that channel. Consequently, when receiving, the receive action may receive only a single value (if there was only one participant in the role containing the dual endpoint of the channel), or multiple values. We did not wish to specify sets or lists of values within our language, so instead, our receive construct has two continuations, the first of which (indicated in square brackets) is invoked on all but the last value of the receive action, whilst the second continuation is invoked on the very last value. This allows the continuations to be invoked as soon as possible and prevents having to wait for all the participants within the sending role to send their messages.

For the auctioneer, the same continuation is specified in both cases (i.e. many bidders may be within the *Bidders* role at the same time and so are placing bids as part of the same send action): the bid that is received is checked to see if it is higher than the current price, and if so, the bid is announced. Eventually, the auctioneer will notice that the auction has expired, will reannounce the winning bid, and will exit.

3 The language

The auction example in the previous section makes use of all the syntax of our language. The metavariable `C` ranges over conversation names, `r` ranges over role names, `k` ranges over channel names and `y` ranges over conversation variables. We write $\overline{\texttt{Rdec}}$ as a shorthand for $\texttt{Rdec}_1 \ldots \texttt{Rdec}_n$ (and similarly for `Kdec`). We abbreviate $\texttt{k}_1 : \texttt{T}_1, \ldots, \texttt{k}_n : \texttt{T}_n$ by writing $\overline{\texttt{k}} : \overline{\texttt{T}}$.

Cdec	::=	conversation C $\{\overline{\texttt{Rdec}}, \overline{\texttt{Kdec}}\}$	Conversation declaration
Rdec	::=	role r	Role declaration
Kdec	::=	channel k k′ (r,r′):T	Channel declaration
P	::=	new C(y); P	New conversation
	|	join(y.r); P	Join
	|	leave(y.r); P	Leave
	|	y.k!(e); P	Send
	|	y.k?(x)[P]P	Receive
	|	P‖P	Parallel composition
	|	μX.P	Recursion
	|	X	Process variable
	|	**0**	Inaction
e	::=	x | y | v | ...	Expression

The declaration conversation C $\{\overline{\texttt{Rdec}}, \overline{\texttt{Kdec}}\}$ defines a conversation of name C and contains the declarations of the roles that take part in the conversation and the declarations of the channels used by the roles within the conversation to communicate. A role declaration of the form role r defines a role of name r. A channel declaration of the form channel k k′ (r,r′):T specifies that the channel is shared by the roles r (with local name k) and r′ (with local name k'). T will be the session type of k at r and the dual of T will be the type of k′ at r′. The session type T must be contractive and is defined by the grammar $\texttt{T} ::= !(\texttt{t}).\texttt{T} \mid ?(\texttt{t}).\texttt{T} \mid \texttt{end} \mid \mu Y.\texttt{T} \mid Y.\texttt{T}$ where Y ranges over session type recursion variables. The dual of a session type is obtained by syntactically substituting ! with ? in the standard way.

Joining a role enables a process to use the channels that are connected to the role as specified by the conversation. We wish to ensure that all participants within a role receive the same set of messages within each receive action. This means that on the sending side, it must not be possible for a participant to join a role and then send a message as part of a send action for which other participants within that role

have already sent. If this were permitted then participants receiving these messages would potentially receive different messages depending on when they see what they each believe is the last message within the receive action. As such, when joining a role, the participant may only send messages as part of send actions which have yet to be started by any participant within the role.

Similarly, to avoid having to preserve message queues indefinitely, when joining a role, the participant may only participate in receive actions that are either yet to start, or have started but have not yet been fully received by all participants within the role. The operational semantics for join uses the inferred types of the channels within the role as used by the process, ensures these are reductions of the declared types of the roles as defined by the conversation, and only reduces when the runtime types of the channels matches the inferred used of the channels.

The process $\texttt{new}\,\texttt{C}(\texttt{y});\texttt{P}$ creates a new instance of the conversation C and continues as the process P in which the local conversation variable y is substituted by the new instance. The process $\texttt{join}(\texttt{y.r});\texttt{P}$ joins the role r within the conversation y. The process $\texttt{leave}(\texttt{y.r});\texttt{P}$ leaves the role r within the conversation y, loses the right to access the channels of y and then continues as P. Sending and receiving are the only two actions that are specified in the session type of a channel.

Sending is asynchronous, and the receive will start as soon as any of the values to be sent as part of the sending action are received. The process $\texttt{y.k!}(\texttt{e});\texttt{P}$ sends the result of the evaluation of e on the channel k of the conversation y and continues as P. The process $\texttt{y.k?}(\texttt{x})[\texttt{P}']\texttt{P}$ receives a value from all the processes in the role of the dual of the channel k, performing $\texttt{P}'$ on all but the last value, and continuing with P on the final value to be received. The parallel composition, recursion and inaction are standard. An expression e could be a value, a variable or other expression coming from the host language.

4 Related work and Conclusions

Session types were first presented by Honda, Vasconcelos and Kubo [8] and have received a great deal of subsequent study and refinement [12, 2, 6, 14, 13, 7, 4, 5, 10].

In these subsequent papers, constructs to support sessions were added to different several languages, and despite session types being able to guarantee many desirable properties (in some cases including progress, see [6, 5] among others), those languages were not able to represent more complex models of interaction at a sufficiently high level to permit an intuitive overview of the communication patterns. To overcome this limitation, multiparty sessions [9, 1, 3] have been proposed.

The existing work on multiparty sessions was the inspiration for us to model interactions between roles instead of processes, with the idea to describe those scenarios in which many participants follow the same communication behaviour (e.g. the bidders in the auction). We believe that our conversations (in the spirit of [3, 9]) make it easy to reason about the overall communication among roles.

In order to maximise potential concurrency, we invested much effort to ensure that processes within a role do not have to operate within lockstep but can each manipulate channels freely at their own rate, provided they do not violate the constraints placed upon their actions by the session types of the channels. This increases the flexibility of the semantics and allows greater levels of parallelism to be exploited. The trade off is that joining and leaving a role dynamically becomes more tricky to define safely.

The flexibility afforded by participants being able to join and leave roles has allowed us to simplify the standard session type semantics. We no longer explicitly cater for delegation because it can be simulated by a participant leaving a role and then the target participant joining the same role, thus gaining access to the role's channels. Similarly, branching can be encoded through changing roles, although we are continuing to explore including branching directly within our language. We are currently in the process of developing a type system to enfore the safety requirements of our language and hope to present this in future work.

Acknowledgements We thank Professor Mariangiola Dezani for making this work possible.

References

[1] Eduardo Bonelli and Adriana Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2007.

[2] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. Funct. Progr.*, 15(2):219–248, 2005.

[3] Luis Caires and Hugo Torres Vieira. Conversation Types. In *ESOP'09*, 2009. To appear.

[4] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[5] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[6] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[7] Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. Bass: Boxed ambients with safe sessions. In *PPDP'06*, pages 61–72. ACM Press, 2006.

[8] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*. Springer, 1998.

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

[10] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008*, pages 516–541. Springer, 2008.

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

[12] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. In *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.

[13] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theorical Computer Science*, 368(1-2):64–87, 2006.

[14] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisited. In *SecRet'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.