Certifying Data in Multiparty Session Types

Bernardo Toninho and Nobuko Yoshida*

Imperial College London
London, United Kingdom

Abstract. Multiparty session types (MPST) are a typing discipline for ensuring the coordination and orchestration of multi-agent communication in concurrent and distributed programs. However, by mostly focusing on the communication aspects of concurrency, MPST are often unable to capture important data invariants in programs. In this work we propose to increase the expressiveness of MPST by considering a notion of value dependencies in order to certify invariants of exchanged data in concurrent and distributed settings.

1 Introduction

Theoretical principles can have transformational effects on computing practice. Well-known examples include program logics and the structured programming discipline. Many theoretical principles established by Philip Wadler have already produced a broad impact on current practices. Wadler's work was instrumental in the introduction of generic types to Java [10], which are now an established feature of statically typed languages such as Java, C#, and the .NET framework. He was a co-designer of Haskell, and features he designed have influenced a wide range of programming languages such as F# and Scala; and database languages such as Ferry and LINQ.

At the core of Wadler's long list of contributions is the notion of types as the fundamental tool for abstraction and reasoning about programs, and as a means of exposing a program's true meaning. In more recent work, Phil has devoted some his efforts to tackling the challenges of communication, concurrency and distributed computation. Naturally, and fruitfully, the answer presents itself in type form.

Meeting these challenges, our mobility group [13] is working with Wadler within the scope of our EPSRC project, *ABCD: A Basis for Concurrency and Distribution* [1]. We quote from *Ambition and Vision* which was (mainly) written by Wadler:

Ambition and Vision [1]. *The data type is one of computing's most successful concepts. The notion of data type appears in programming languages from the oldest to the newest, and it covers concepts ranging from a single bit to organised tables containing petabytes of data. Types act as the fundamental unit of compositionality: the first thing a programmer writes or reads about each method or module is the data types it acts upon, and type discipline guarantees that each call of a method matches its definition and each import of a module matches its export. Data types play a central role in all*

* This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU FetOpen UPSCALE 612985.

aspects of software, from architectural design to interactive development environments to efficient compilation.

The ambition of our project is to position *session types* as the analogue of the data type for concurrency and distribution. Session types impose structure to sessions, in the same way data types impose structure to data instances. Session types were first devised two decades ago by Takeuchi, Honda, Kubo and Vasconcelos [17,8], and later developed by Wadler and others [20,21,12,3,4]. Session types build upon data types, as data types specify the lowest level of data exchange, upon which more complex protocols are built. Just as data type discipline matches use and definition of a method, and import and export of a module, a session type discipline ensures consistency and compatibility between the two ends of a communication. Session types offer a mechanism for ensuring communication safety (and a variety of other fundamental properties such as absence of deadlocks or races) of systems involving two interacting parties.

A more general view of a session is that it combines *multiple* interactions forming a meaningful scenario into a single logical unit, offering a basic programming abstraction for communicating processes. Given that in a wide range of application scenarios it is often necessary to specify and ensure the coordination of multiple communicating agents, Honda et al. [9,5] introduced *multiparty* session types (MPST), enabling the specification of interactions involving multiple peers from a global perspective, which is then automatically mapped (or *projected*) to *local* types that may be checked against the individual endpoint processes. Using this framework, communication safety is ensured among multiple endpoints.

This paper seeks to extend Wadler's viewpoint of a session type: we propose a session type discipline for expressing and certifying global properties that may depend on the exchanged data, by introducing value dependent types for multiparty sessions.

Our proposed typing discipline ensures that implementations of a multiparty conversation not only adhere to the session discipline but also satisfy rich constraints imposed on the exchanged data, which may be explicitly witnessed at runtime by *proof objects*. Our aim is to thus raise the standard of types in concurrency to that of data types: the programmer with a precise description of the interaction patterns followed by the communicating parties but also specify (and *certify*) the global invariants of data that are required and ensured throughout the multiparty communication.

2 Multiparty Session Types and Certified Data

This section motivates a technique to certify properties of exchanged data in multiparty session types (MPST) through a notion of value dependencies [19]. We begin with a brief introduction of the original MPST framework and its shortcomings with respect to expressing certain functional constraints on global protocols. We then address these issues through the use of value dependencies in the framework.

In MPST, we begin with a *global type*, consisting of a global view of the interactions shared amongst the several interested parties. For instance, the following consists of a global type specification of a toy protocol involving three parties:

$$G = \mathsf{p} \to \mathsf{q} : (\mathsf{Int}).\mathsf{p} \to \mathsf{r} : (\mathsf{String}).\mathsf{q} \to \mathsf{r} : \left(\mathsf{yes}{:}G'; \mathsf{no}{:}\mathbf{end}\right) \tag{1}$$

$$
\begin{aligned}
G \upharpoonright \mathsf{p} &= \mathsf{q}!(\mathsf{Int}); \mathsf{r}!(\mathsf{String}); \mathbf{end} \\
G \upharpoonright \mathsf{q} &= \mathsf{p}?(\mathsf{Int}); \oplus \mathsf{r}(\mathsf{yes} : G' \upharpoonright \mathsf{q}; \mathsf{no} : \mathbf{end}) \\
G \upharpoonright \mathsf{r} &= \mathsf{p}?(\mathsf{String}); \& \mathsf{q}(\mathsf{yes} : G' \upharpoonright \mathsf{r}; \mathsf{no} : \mathbf{end})
\end{aligned}
$$

Fig. 1. Projections of Global Type G (with $\mathsf{p} \notin G'$)

In the specification G above, participant p sends an integer and a string to participants q and r, respectively. Afterwards, q will either send to r a no message, ending the global interaction; or a yes message, causing the interaction to proceed to G'. We may assume that G specifies a portion of some coordinated agreement, such as that between a service broker p, giving a price quote to a client q while making a tentative reservation of the service to provider s (encoded as a string), which is then accepted or rejected by q.

Given a global type, we must define a notion of *projection*, which constructs the view for each endpoint of the global interaction as a *local type*. For G, the projection of G for p, written $G \upharpoonright \mathsf{p}$, is given by the local type $\mathsf{q}!(\mathsf{Int}); \mathsf{r}!(\mathsf{String}); \mathbf{end}$, assuming p does not participate in G', which describes the parts of the global interaction that pertain to actions of participant p. Given the projected local types (Fig. 1) for each communicating party we may then check that the global specification is satisfied by the interactions of the several endpoint processes to ensure deadlock-freedom.

The framework sketched above is only suited for describing the shape of communication. While we may argue that G does indeed specify the interactions between a service broker p, a client q, and a provider r, such a global specification is satisfied by many process instances of p, q and r that may not in fact offer the desired functionality. For instance, the implementation of the broker p may send an incorrect price to client q, or the wrong service identifier to provider r and the system would still be correct according to G. The crucial issue is that while a global type specifies precisely *how* parties communicate, it only captures *what* the parties should communicate in a very loose sense (for example, "send a string" vs. "send a string corresponding to the service code for which the client was sent the price").

To overcome this issue, we propose the adoption of *value dependent* multiparty session types, which refine multiparty session types by adding type dependencies to specifications of exchanged data (extending the work of [19,15] for the binary setting).

The technical challenge here is reconciling the global specification of the distributed interaction, which may reference properties depending on data spread across multiple endpoints, with the local knowledge of each participant. Projection must ensure that whenever two endpoint processes exchange a proof object, the object is consistent with the knowledge of both endpoints. Specifically, the sender must know each term referenced by the proof object and propagate the relevant information to the receiver in a consistent way. Another issue is that the local types generated for each endpoint may not necessarily have matching dependencies due to the potentially incomplete views of the global agreement (for instance, participant p may assert some relationship between two data elements to q, where q only knows one of the datum). We must nevertheless ensure that endpoint projections are well-formed given the local knowledge of each endpoint and preserve the intended data dependencies, given the partial view of the system.

2.1 Value Dependent Multiparty Session Types

The key motivation for using value dependent types is to enable type level specifications of properties of data used in computation. Given that MPST have a natural distributed interpretation, we wish not only to express properties of exchanged data but also to support the ability for processes to exchange *proof objects* witnessing the properties of interest, providing a degree of certified communication in some sense. For instance, a value dependent version of G above can be:

$$\begin{aligned} G_{Dep} = \mathsf{p} \to \mathsf{q} : (x{:}\mathsf{Int}).\mathsf{p} \to \mathsf{r} : (y{:}\mathsf{String}). \\ \mathsf{p} \to \mathsf{q} : (z{:}\mathsf{isPrice}(x, y)).\mathsf{q} \to \mathsf{r} : \left(\mathsf{yes}{:}G'; \mathsf{no}{:}\mathbf{end}\right) \end{aligned} \tag{2}$$

where the predicate $\mathsf{isPrice}(x, y)$ holds only if the integer x is indeed the price for service y. In the specification G_{Dep}, p must also send to q a *proof* of the relationship between the previously sent price and the service code. While the notion of proofs as first-class objects might seem somewhat foreign insofar as one might simply expect some runtime verification mechanism that ensures the received data is in the required form (as specified by the type-level *assertions*), explicit proof exchange is a more general approach: proof generation might not be decidable in general, whereas proof checking should be. Moreover, even when proof generation is decidable, it can often require more computational resources than checking the validity of a proof object.

By leveraging the Curry-Howard correspondence between propositions and types (and proofs and programs), we can represent such proof objects as terms in a language with a suitable (dependent) type discipline, such that the only well-typed instances of processes implementing the role of p will be those that not only adhere to the session discipline but also satisfy the functional constraints encoded in the dependently typed values. The framework also ensures that proof objects are explicitly exchanged between communicating parties, which is of practical significance in a distributed setting.

Global Types The syntax for value dependent MPST is given in Figure 2. A message exchange $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G$ specifies communication between sender p and receiver q of a value of type τ, bound to x in G. The type structure of τ is somewhat generic, with the following requirements: we assume a dependently typed λ-calculus with dependent functions $\Pi x{:}\tau.\sigma$ and pairs $\Sigma x{:}\tau.\sigma$, where x binds its occurrence in Π and Σ. In our theory and examples, we generalise dependent pair types $\Sigma x{:}\tau.\sigma$ to $\Sigma l.\sigma$, where l is a *list* of type bindings of the form $x_i{:}\tau_i$. We manipulate such lists using Haskell-style notation. We assume some base types b and singleton types [16], written $\mathcal{S}(M)$, where M is a value of some base type b and $\mathcal{S}(M)$ denotes a value of type b equal to M. For example, if we assume natural numbers Nat as base types then the natural number 5 can be typed with both Nat and $\mathsf{Nat}(5)$. We require type preservation and progress for this language of message values, as well as decidability of type-checking (although, crucially, not of type inhabitance).

The branching $\mathsf{p} \to \mathsf{q} : \left(l_j{:}G_j\right)_{j \in J}$ denotes a selection made by p between a set of behaviours G_j identified by labels l_j, achieved by the emission of a label l_i with $\{\mathsf{l}_i : i \in J\}$ from p to q. The session then continues as G_i for all participants.

Session delegation $\mathsf{p} \to \mathsf{q} : (T).G$ denotes that participant p delegates to q its interactions with a session channel of local type T (defined below), achieved by sending

$$
\begin{array}{rl}
G & ::= \mathsf{p} \to \mathsf{q} : (x{:}\tau).G \\
& \mid \mathsf{p} \to \mathsf{q} : \left(\mathsf{l}_j{:}G_j\right)_{j\in J} \\
& \mid \mathsf{p} \to \mathsf{q} : (T).G \\
& \mid \mu t\ (x = M{:}\tau).G \mid t\langle M\rangle \\
& \mid \mathbf{end}
\end{array}
\qquad
\begin{array}{rll}
T, U & ::= \mathsf{p}!(x{:}\tau); T & \mid \mathsf{p}?(x{:}\tau); T \\
& \mid \oplus\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I} & \mid \&\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I} \\
& \mid \mathsf{p}!(U); T & \mid \mathsf{p}?(U); T \\
& \mid \mu t\ (x = M{:}\tau).T & \mid t\langle M\rangle \\
& \mid \mathbf{end} &
\end{array}
$$

$$\tau, \sigma ::= \Pi x{:}\tau.\sigma \mid \Sigma x{:}\tau.\sigma \mid b \mid \mathcal{S}(M)$$

Fig. 2. Syntax of Global and Local Types

the channel endpoint, after which the interaction proceeds as G. Note that there is no binding for T since we only consider dependencies of *values*, rather than on sessions.

Recursive global types $\mu t\ (x = M{:}\tau).G$, where t and x bind its occurrences in G, enable the specification of how a recursive interaction should proceed among the different participants. The parameter x is a recursion variable standing for a term M of type τ, which defines the initial value of x in the first recursive instance, acting as a parameter of the recursion. A recursion is instantiated with $t\langle M\rangle$, where M denotes the value taken by x in the next instance. We assume that recursive type definitions are contractive and, for the sake of simplicity, that there is at least one occurrence of t in G. We consider recursive types in the typical equirecursive sense, up to unfolding. Finally, $\mathbf{end}$ denotes a lack of further interactions. We often omit $\mathbf{end}$ and write message exchange and recursion as $\mathsf{p} \to \mathsf{q} : (\tau).G$ and $\mu t.G$ if x does not occur in G.

We write $fv(G)$ for the free variables of G, defined inductively in the usual way. We state that a global type G is *closed* (resp. *open*) if $fv(G) = \emptyset$ (resp. $fv(G) \neq \emptyset$). We write $\mathbb{C}[\![-]\!]$ for a global type context (i.e., a global type with a hole). $G \sqsubseteq G'$ stands that G is a subterm of G'.

Definition 2.1 (Global Type Context). *Given a global type, we define its subterms via the following notion of context:*

$$\mathbb{C} ::= \_ \mid \mathsf{p} \to \mathsf{r} : (x{:}\tau).\mathbb{C} \mid \mathsf{p} \to \mathsf{r} : (T).\mathbb{C} \mid \mathsf{p} \to \mathsf{r} : \left(l_i : \mathbb{C}_i\right)_{i\in I} \mid \mu t(x = M{:}\tau).\mathbb{C} \mid \mathbf{end} \mid t\langle M\rangle$$

with the hole $\_$ *occurring in at most one* $\mathbb{C}_i$.

Local (endpoint) Types Value dependent *local types* specify the behaviour and data constraints of each endpoint involved in the multiparty session. The send types $\mathsf{p}!(x{:}\tau); T$ and $\mathsf{p}!(U); T$ denote, respectively, sending a value M of type τ to participant p and proceeding with the behaviour $T\{M/x\}$ or sending a channel of type U and continuing with behaviour T. The selection type $\oplus\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I}$ encodes the transmission to p of a label l_i following by the communication specified in T_i. Receive types ($\mathsf{p}?(x{:}\tau); T$ and $\mathsf{p}?(U); T$) and branch types $\oplus\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I}$ specify the dual behaviours of sending and selection. Recursive types $\mu t\ (x = M{:}\tau).T$ (and their instantiations $t\langle M\rangle$) specify a recursive behaviour T parameterised by a term M of type τ, bound to x.

$$
\begin{aligned}
G_{BSD} \triangleq\ & \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{query} : \mathsf{Nat}).\\
& \mathsf{Distr} \to \mathsf{Seller} : (\mathsf{stock} : \mathsf{Int}).\\
& \mathsf{Seller} \to \mathsf{Buyer} : (\mathsf{q} : (\mathsf{Int}(\mathsf{stock}), \mathsf{Double})).\\
& \mathsf{Buyer} \to \mathsf{Seller} : (\mathsf{ok} : \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{ok} : G_{ok}),\\
& \qquad\qquad \mathsf{quit} : \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{quit} : \mathbf{end})\\
G_{ok} \triangleq\ & \mu t(x = \langle \pi_2(\mathsf{q}), \mathsf{inl\ refl}\rangle : \Sigma y : \mathsf{Double}(\pi_2(\mathsf{q})).y \geq \pi_2(\mathsf{q})).\\
& \mathsf{Buyer} \to \mathsf{Seller} : (\mathsf{offer} : \Sigma z : \mathsf{Double}.z \geq \pi_2(\mathsf{q})).\\
& \mathsf{Seller} \to \mathsf{Buyer} : (\mathsf{hag} : G_{hag}, \mathsf{exit} : \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{cancel} : \mathbf{end}),\\
& \qquad\qquad \mathsf{sell} : \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{commit} : \mathbf{end}))\\
G_{hag} \triangleq\ & \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{hag} : t\langle \pi_1(\mathsf{offer}), \pi_2(\mathsf{offer})\rangle)
\end{aligned}
$$

Fig. 3. Buyer - Seller - Distributor Global Type

2.2 Examples of Value Dependent Global Types

We now introduce two examples of value dependent global types, showcasing their heightened expressiveness.

Three party interaction: Buyer - Seller - Distributor We specify the interaction patterns between three parties: a buyer, a seller and a distributor, illustrating the combined use of recursion and dependencies (Figure 3).

The session begins with the buyer requesting a query of a product from the distributor. The distributor then communicates with the seller, sending the number of items currently available. The seller sends to the buyer the number of available product and an initial price. The buyer and the seller then initiate in a recursive negotiation, where the buyer selects to either proceed with the negotiation or to quit the protocol. In the latter case, the seller notifies the distributor of the cancellation. In the former, the buyer sends the seller an offer, upon which the seller must decide whether to continue negotiating, to terminate the negotiation by rejecting the offer, or to terminate the negotiation by accepting the offer. The decision is then forwarded to the distributor.

This interaction, beyond the equality constraints between the stock message sent from the distributor to the seller and then from the seller to the buyer, captures in a relatively simple way the encoding of the loop invariant – that each offer made by the buyer is always increasing, and at least as much as the initial quote.

MapReduce We specify a distributed computation where a client sends to a server some data upon which the server is intended to run some potentially computationally expensive computation, represented by a map-style function f and a reduce-style function g.

$$
\begin{aligned}
G_{MR} \triangleq\ & \mathsf{Client} \to \mathsf{Server} : (\mathsf{d} : \mathsf{String}).\\
& \mathsf{Server} \to \mathsf{Worker}_1 : (\mathsf{d}_1 : \mathsf{String}).\mathsf{Server} \to \mathsf{Worker}_2 : (\mathsf{d}_2 : \mathsf{String}).\\
& \mathsf{Server} \to \mathsf{Aggr} : (\mathsf{p} : \mathsf{d} = \mathsf{d}_1 +\!\!+ \mathsf{d}_2).\\
& \mathsf{Worker}_1 \to \mathsf{Aggr} : (\mathsf{r}_1 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1)).\\
& \mathsf{Worker}_2 \to \mathsf{Aggr} : (\mathsf{r}_2 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2)).\\
& \mathsf{Aggr} \to \mathsf{Server} : (\mathsf{r}_3 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)))\\
& \mathsf{Server} \to \mathsf{Client} : (\mathsf{res} : \mathsf{String}(\pi_1(\mathsf{r}_3)))
\end{aligned}
$$

Upon receiving the data from the client, the server divides it into two parts which are then sent to be processed by the two workers. The system includes an aggregator service, which is informed by the server of the division of the data. The workers then send to the aggregator the result of the computation f on their respective data partitions, which then sends back to the server the aggregation result (computed using the aggregation function g). Finally, the server sends back to the client the final result.

The crucial aspect of this simple example is that not only are we describing the structure of communication (and to some extent, the topology of the service), we are specifying in a very precise way the actual functionality of the global coordination.

Recursive Game To clarify the interaction of recursion and value dependencies, we encode a simple toy game protocol between three parties: Alice, Bob and Carol.

$$
\begin{aligned}
G_{ABC} \triangleq\ & \mathsf{Carol} \to \mathsf{Alice} : (n : \Sigma y{:}\mathsf{Nat}.y > 0).\\
& \mathsf{Carol} \to \mathsf{Bob} : (n' : \mathsf{Nat}(\pi_1(n))).\\
& \mu t(x = n : \Sigma y{:}\mathsf{Nat}.y > 0).\\
& \mathsf{Alice} \to \mathsf{Carol} : (m : \mathsf{Nat}).\\
& \mathsf{Bob} \to \mathsf{Carol} : (m' : \mathsf{Nat}).\\
& \mathsf{Carol} \to \mathsf{Alice} : (\mathsf{correct} : G_{c1}, \mathsf{wrong} : t\langle x - 1, M\rangle)\\
G_{c1} \triangleq\ & \mathsf{Carol} \to \mathsf{Bob}(\mathsf{correct} : \mathsf{end}, \mathsf{wrong} : t\langle x - 1, M\rangle)
\end{aligned}
$$

In the protocol above, Carol sends both Alice and Bob a number of total tries n the two participants are allowed to attempt to guess some random number generated by Carol. The protocol then proceeds by repeatedly accepting guesses from both Alice and Bob until they both guess correctly, upon which the protocol terminates, or until the number of tries n runs out.

While very minimal in its features, this example showcases how the combination of recursion and value dependencies allows us to specify sophisticated global types, such as counting down from a sent or received number, insofar as we are able to make the actual communication structure of the protocol depend on previously received data.

2.3 Well-formedness of Global Types

We detail the well-formedness conditions on global value-dependent MPST. In contrast with the work on design-by-contract [2], which introduces assertions to MPST, we do not in general enforce the property that all well-formed global types are realisable by some well-typed endpoint processes. In [2], global type well-formedness entails that assertions expressed in a global type are possible to satisfy, by restricting the assertion language to decidable logics. Given our aim of maintaining a general dependent type theory as our proof language, we opt for a different design.

In our general setting, we can use a larger set of well-formed global types for which no process realisers may exist. The decision problem of determining if such process realisers exist is itself undecidable. Our goal is to define well-formedness of global and local types such that:

1. Projection of a well-formed global type produces well-formed local types by a simple projection rule; and

2. If a collection of processes which satisfy local types exist, then the global specification is satisfied.

Below we define simple *well-formedness conditions* which are sufficient to ensure the above properties. The first condition defines a binding restriction on recursions; the second captures the fact that in a message exchange between two participants, the sender should always know all the message variables mentioned in the message's type. We note that history-sensitivity has been shown decidable in [2], where a compositional proof system for history sensitivity is presented.

Definition 2.2 (Well-formedness conditions).

1. (*recursion*) Let G be a closed global type. We say that G has well-formed recursion iff for all $t\langle M\rangle \in G, x \in fv(M)$, there exists $\mathbb{C}[\![-]\!]$ such that either: $G = \mathbb{C}[\![\mathsf{p} \to \mathsf{q} : (x{:}\sigma).G']\!]$ or $G = \mathbb{C}[\![\mu t\ (x = M : \tau).G']\!]$.
2. (*history sensitivity*) Given a global type G, we say that p *ensures* τ in G iff there is $\mathbb{C}$ such that $G = \mathbb{C}[\![\mathsf{p} \to \mathsf{q} : (x{:}\tau).G']\!]$. Then for any natural number n, G is n-history sensitive on a message variable x iff for all G' such that G' is a n-times unfolding of G, and for all types τ in G' such that $x \in fv(\tau)$ there is $\mathsf{p} \to \mathsf{q} : (x{:}\tau').G'' \sqsubseteq G'$ such that p or q ensures τ in G''. We say that G is *history sensitive* iff it is n-history sensitive for all natural numbers n on all message variables in G.

G is well-formed if all recursions in G are well-formed and G is history sensitive.

Hereafter we consider only well-formed global types.

3 Projection and Data Dependencies

We now motivate some of the challenges of defining projection of a global type while respecting the partial local knowledge of each participant.

Recall the global type G_{Dep} of Section 2.1 (Equation 2), which is a well-formed global type. In the final interaction between participants p and q, p is supposed to send a proof that the previously received integer value x is indeed the price for the service code sent to r, identified by the string y. From the perspective of p, the value of both x and y are known. However, q only knows the value of x since y was sent only to r. Thus, if we consider a typical notion of projection that traverses the type G_{Dep} and collects the direction of communication accordingly we obtain the following local types for p and q (for some T_{q}):

$$G_{Dep} \upharpoonright \mathsf{p} = \mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\mathsf{isPrice}(x, y)); \mathbf{end}$$
$$G_{Dep} \upharpoonright \mathsf{q} = \mathsf{p}?(x{:}\mathsf{Int}); \mathsf{p}?(z{:}\mathsf{isPrice}(x, y)); T_{\mathsf{q}}$$

The local type for q cannot be correct since it contains a free variable y (given q's local knowledge), which is not free in the local type for p. In order to generate adequate local types for both endpoints we must ensure that the two types respect the local knowledge of each participant.

Intuitively, the type for the endpoint corresponding to participant p must bundle in the message identified by z all the unknown information from participant q's perspective. However, if we modify the projection for participant p to,

$$\mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\Sigma y'{:}\mathsf{String}.\mathsf{isPrice}(x, y')); \mathbf{end} \tag{3}$$

we do not preserve the semantics of G_{Dep}, in the sense that a process with the type above may send to q *any* price, provided it is indeed the price of a service in the system.

In order to preserve both the semantics of data dependencies in global types *and* generate well-formed local types for both endpoints, we make use of singleton types and subtyping, which is formally defined in Section 3.1. Crucially, we make use of singleton types to implicitly refer to the equality constraints induced by dependencies in a global type. In the example above, generating the following local type for p,

$$\mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y')); \mathbf{end} \tag{4}$$

we can preserve the semantics of G_{Dep}, in the sense that p may only send x and y such that one is the price of the other. Moreover, we exploit the fact that for any base type b, if $M : b$ then $\mathcal{S}(M) \leq b$ in order to produce the following local type for endpoint q,

$$\mathsf{p}?(x{:}\mathsf{Int}); \mathsf{p}?(z{:}\Sigma y'{:}\mathsf{String}.\mathsf{isPrice}(x, y')); T_{\mathsf{q}} \tag{5}$$

The type above not only respects the local knowledge of endpoint q but is also compatible with the interactions specified by the local type for endpoint p due to the subtyping of singletons, since $\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y') \leq \Sigma y'{:}\mathsf{String}.\mathsf{isPrice}(x, y')$ by the usual covariant subtyping rules for Σ-types and the fact that a singleton is always a subtype of its corresponding base type (we note that session subtyping for message input is covariant in the message type; and dually, contravariant for output).

3.1 Defining Projection

Having discussed the main challenges of preserving global data dependencies in local types, we define a notion of projection that generates *compatible* message types (in the sense of Def. 3.2) for well-formed global types.

We begin by introducing the subtyping rules for both local and data types. The rules are mostly standard from the literature of subtyping in session types [7] and singleton types [16]. For conciseness we only consider session subtyping for input and output types. Subtyping for choices and branching are orthogonal. The subtyping judgement, written $\Psi \vdash \tau \leq \sigma$ for data types and $\Psi \vdash T \leq S$ for local types, denotes that τ (resp. T) is a subtype of σ (resp. S), where Ψ is a context tracking free variables in types. Note that if T is a subtype of U, then a process implementing type T may be safely used wherever one of type U is expected. We write $\Psi \vdash M : \tau$ for the typing judgement of terms M, which we maintain mostly unspecified. We write $\Psi \vdash \tau$ for the well-formedness of τ and $\Psi \vdash M \equiv N : \tau$ for definitional equality of M and N. The key subtyping rules are given in Figure 4.

The key rules for the development of a well-defined notion of projection is the singleton subtyping rule (SUB-$\mathcal{S}$), which specifies that a singleton for a base type is always

$$\frac{\Psi \vdash M : b}{\Psi \vdash \mathcal{S}(M) \leq b}\ (\text{SUB-}\mathcal{S}) \qquad \frac{\Psi \vdash M_1 \equiv M_2 : b}{\Psi \vdash \mathcal{S}(M_1) \leq \mathcal{S}(M_2)}\ (\text{SUBEQ-}\mathcal{S})$$

$$\frac{\Psi \vdash \Pi x{:}\tau_1'.\tau_1'' \quad \Psi \vdash \tau_2' \leq \tau_1' \qquad \Psi, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''}{\Psi \vdash \Pi x{:}\tau_1'.\tau_1'' \leq \Pi x{:}\tau_2'.\tau_2''}\ (\text{SUB-}\Pi) \qquad \frac{\Psi \vdash \Sigma x{:}\tau_1'.\tau_1'' \quad \Psi \vdash \tau_1' \leq \tau_2' \qquad \Psi, x{:}\tau_1' \vdash \tau_1'' \leq \tau_2''}{\Psi \vdash \Sigma x{:}\tau_1'.\tau_1'' \leq \Sigma x{:}\tau_2'.\tau_2''}\ (\text{SUB-}\Sigma)$$

$$\frac{\Psi \vdash M \equiv N : b}{\Psi \vdash \mathcal{S}(M) \equiv \mathcal{S}(N)}\ (\text{TEQ-}\mathcal{S}) \qquad \frac{\Psi \vdash M \equiv N : \sigma \qquad \Psi \vdash \sigma \leq \tau}{\Psi \vdash M \equiv N : \tau}\ (\text{EQ-}\leq)$$

$$\frac{\Psi \vdash \tau \leq \tau' \qquad \Psi, x{:}\tau' \vdash T \leq T'}{\Psi \vdash \mathsf{p}?(x{:}\tau).T \leq \mathsf{p}?(x{:}\tau').T'}\ (\text{SUB-}?) \qquad \frac{\Psi \vdash \tau' \leq \tau \qquad \Psi, x{:}\tau \vdash T \leq T'}{\Psi \vdash \mathsf{p}!(x{:}\tau).T \leq \mathsf{p}!(x{:}\tau').T'}\ (\text{SUB-}!)$$

Fig. 4. Subtyping for Local and Data Types (Abridged)

a subtype of its base type and the rules for subtyping of input and output local types, enabling receiving processes to receive instances of the singleton type when expecting to receive instances of the corresponding base types.

We make precise the notion of a participant knowing the identity of a message or recursion variable occurring in a global type. Intuitively, a participant knows the identity of a message variable if it is involved in corresponding communication. Similarly, knowing a recursion variable requires knowledge of all message variables that occur in the recursive parameter.

Definition 3.1 (Knowledge). Let G be a closed global type and $\mathsf{p} \in G$. We say that p knows $x{:}\tau$ in G iff there is $\mathbb{C}$ such that either:

- $G = \mathbb{C}[\![\mathsf{s} \to \mathsf{r} : (x : \tau).G']\!]$ with $\mathsf{p} \in \{\mathsf{s}, \mathsf{r}\}$; or
- $G = \mathbb{C}[\![\mu t(x = M : \tau).G']\!]$ where for all $y \in fv(M) \cup \bigcup_{t\langle M' \rangle \in G'} fv(M') \setminus \{x\}$ and p knows y in G.

We say that a participant p knows M in G iff p knows all the free variables of M in G.

Equipped with our notion of subtyping and knowledge, we define compatibility between message types in Definition 3.2, appealing to a consistent *priming* of the variables in a type. Given a variable $x{:}\tau$ with $x \in fv(\sigma)$, we say that x' is a primed version of x iff $x'{:}\tau(x)$. A priming of type σ is a pointwise priming of (some) of its free variables. We maintain the connection between a primed variable and its unprimed version (we write $primedVars(r)$ to denote the primed variables of set r).

Definition 3.2 (Compatible Message Types). *Given a well-formed global type G with $\mathsf{p} \to \mathsf{q} : (x : \tau).G' \sqsubseteq G$ we say that the pair of data types (σ_1, σ_2) is compatible with p and q for message x iff*

1. *$\Psi \vdash \sigma_1$ and $\Psi \vdash \sigma_2$, for some Ψ;*
2. *p (resp. q) knows all the (free) variables in σ_1 (resp. σ_2);*

3. $\Psi \vdash \sigma_1 \leq \sigma_2$ *for some* Ψ*;*
4. $\Psi \vdash \sigma_1 \equiv \Sigma l.\tau'$*, for some priming of* τ *and some (possibly empty) list* l*.*

Two message types σ_1 and σ_2 are deemed compatible from the perspective of participants p and q if both types are well-formed, their free variables are known by the corresponding participants and they are related by subtyping. Moreover, we enforce that compatible message types must be dependent tuples (without loss of generality). For example, in the global type G_{Dep} discussed above, for the last message exchange between participants p and q, the pair of message types $\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y')$ and $\Sigma y'{:}\mathsf{String}.\mathsf{isPrice}(x, y')$ is compatible for p and q, respectively.

We make use of an auxiliary function, dubbed *compatible type binding generation* (CTB), that given a message exchange $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G'$ in a global type G produces a dependent tuple $\Sigma l.\tau'$, where τ' is a *priming* of τ (τ and τ' differ only on the names of free variables of base type, where $x' \in fv(\tau')$ corresponds to $x \in fv(\tau)$) and l is a list of variable bindings (occurring in τ') that are known by participant p and not known by q, making use of singleton types to preserve the value dependencies specified in the global type G.

Definition 3.3 (Compatible Type Binding Generation). *For any closed global type* G*, with* $\mathsf{p} \to \mathsf{q} : (x : \tau).G' \sqsubseteq G$*. We generate a compatible type binding for* $x{:}\tau$*, written* $CTB(x{:}\tau)$*, as follows. If* τ *is a base type then* $CTB(x{:}\tau) = [x{:}\tau]$*. Otherwise, the compatible type binding for* $x{:}\tau$ *is given by the recursive function* $F(x{:}\tau)$*, given below making use of typical list manipulation notation:*

1. *If* τ *is a base type, then* $F(x{:}\tau) = [x'{:}\tau(x)]$*; otherwise,*
2. *Let* u *be the list of bindings corresponding to the free variables of* τ*, known by* p *and not by* q*.*
3. *If* $u = [\,]$ *then* $F(x{:}\tau) = [x{:}\tau]$*; otherwise,*
4. *Let* $r = \mathsf{fold}\ (\lambda b.\lambda acc\ .\mathsf{merge}(F(b), acc))\ [\,]\ u_1$*.*
5. *Let* $\tau' = \tau\{primedVars(r)'/primedVars(r)\}$*, then* $F(x{:}\tau) = r +\!\!+ [x{:}\tau']$*.*

We note that in recursive calls to F*, the participants* p *and* q *are fixed in the sense that* $F(b)$ *considers variables in the binding* b *known by* p *and unknown by* q*. Moreover, usages of CTB tacitly assume that we convert the resulting list into a dependent tuple in the natural dependency-preserving way.*

For instance, in the global type G_{Dep} (Equation 2), the CTB for the third exchange between p and q produces the type $\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y')$ which may be used as the message type for the output of p, bundling all the necessary data that is unknown by q at the given point in the protocol. The key insight is that CTB consists of a terminating function that computes a tuple bundling all the unknown information from the perspective of the recipient of a message, using singleton types to preserve data dependencies from the perspective of the sender.

Theorem 3.4 (Compatible Type Binding Generation – Termination). *Compatible type binding generation (Definition 3.3) is a terminating function.*

Proof. We first point out that the fact that we consider *closed* global types enforces some constraints on free variables in data types. In particular, a data type's free variables must have all been defined by *previous* communication actions, which immediately excludes circular dependencies where two data types mutually depend on each other (e.g. the binding for a free variable y of a type $x{:}\mathcal{P}(y)$ being of the form $y{:}\mathcal{Q}(x)$). .

Function $CTB(x{:}\tau)$ in Definition 3.3 inspects bindings of free variables of τ, known by participant p and unknown by q (c.f. Definition 3.1). By construction, these variables are bound by previous interactions involving p. Since there is no possibility for circularity, the free variables of a data type and the free variables of types in their binding occurrences form a *directed acyclic graph* (DAG).

The termination of F follows from the observation that it simply performs a traversal of this DAG, producing a reverse topological ordering of the graph. Specifically, F traverses the subgraph of this DAG made up of variables known by p and not by q (itself a DAG). This is straightforward to see: the terminal nodes of the graph are those when we reach a base type or have no unknown variables; for non-terminal nodes, that is, those with unknown variables, we perform a depth-first search traversal of the DAG, collecting the outcomes in a merged list with the appropriately primed type as the last element. We note that this traversal produces a reverse topological ordering of the DAG. □

To ensure that projection produces well-formed local types for both endpoints, we make use of *singleton erasure* (Definition 3.5) to erase singletons from a dependent tuple. Intuitively, we use CTB to generate the message type for the sender and its singleton erasure to generate the type for the recipient, observing that the two are related by subtyping.

Definition 3.5 (Singleton Erasure). Given a type τ of the form $\Sigma l.\sigma$, we write $\tau^\dagger$ for its singleton erased version, that is, where each primed binding in l of the form $x_i'{:}\sigma_i(x)$ is replaced by $x_i'{:}\sigma_i$.

Finally, for projection of choices and branchings we appeal to a merge operator along the lines of [6], written $T \sqcup T'$, ensuring that if the locally observable behaviour of the local type is not independent of the chosen branch then it is identifiable via a unique choice/branching label (the merge operator is otherwise undefined).

Definition 3.6 (Merge). *Let $T = \&\mathsf{r}\big(\mathsf{l}_i : T_i\big)_{i\in I}$ and $T' = \&\mathsf{r}\big(\mathsf{l}'_j : T'_j\big)_{j\in J}$. The merge $T \sqcup T'$ of T and T' is defined as:*

$$T \sqcup T' \triangleq \&\mathsf{r}\big(\mathsf{l}_h : T_h\big)_{h\in I\setminus J} \cup \big(\mathsf{l}'_h : T'_h\big)_{h\in J\setminus I} \cup \big(\mathsf{l}_h : T_h \sqcup T'_h\big)_{h\in I\cap J}$$
$$T \sqcup T \triangleq T$$

if $\mathsf{l}_h = \mathsf{l}'_h$ for each $h \in I \cap J$. Merge is homomorphic (i.e. $\mathcal{C}[T_1] \sqcup \mathcal{C}[T_2] = \mathcal{C}[T_1 \sqcup T_2]$) and is undefined otherwise.

Definition 3.7 (Global Projection). Let G by a global type. The projection of G in a participant p is defined by the function $G \upharpoonright \mathsf{p}$ in Figure 5. If no side conditions hold then projection is undefined.

$$\mathsf{s} \to \mathsf{r} : (x{:}\tau).G' \upharpoonright \mathsf{p} = \begin{cases} \mathsf{r}!(x{:}(CTB(x{:}\tau))); (G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{s} \\ \mathsf{s}?(x{:}(CTB(x{:}\tau))^{\dagger}); (G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{r} \\ G' \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$\mathsf{s} \to \mathsf{r} : \Big(l_j{:}G_j\Big)_{j \in J} \upharpoonright \mathsf{p} = \begin{cases} \oplus \mathsf{r}\big(l_j : G_j \upharpoonright \mathsf{p}\big)_{j \in J} & \text{if } \mathsf{p} = \mathsf{s} \\ \& \mathsf{s}\big(l_j : G_j \upharpoonright \mathsf{p}\big)_{j \in J} & \text{if } \mathsf{p} = \mathsf{r} \\ \sqcup_{j \in J} G_i \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$\mu t(x = M{:}\tau).G' \upharpoonright \mathsf{p} = \begin{cases} \mu t(x = M{:}\tau).(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \in G' \text{ and } \mathsf{p} \text{ knows } M \\ \mu t.(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \in G' \text{ and } M \text{ unknown to } \mathsf{p} \\ \mathbf{end} & \text{otherwise} \end{cases}$$

$$t\langle M \rangle \upharpoonright \mathsf{p} = \begin{cases} t\langle M \rangle & \text{if } \mathsf{p} \text{ knows } M \\ t & \text{otherwise} \end{cases}$$

$$\mathbf{end} \upharpoonright \mathsf{p} = \mathbf{end}$$

Fig. 5. Projection.

Example 3.8 (MapReduce). A projection of MapReduce in Section 2.2 from the viewpoint of participant Server is given below (projections for the other roles are given in Fig. 6):

$$\begin{aligned} G_{MR} \upharpoonright \mathsf{Server} \triangleq\ & \mathsf{Client}?(\mathsf{d}{:}\mathsf{String}); \mathsf{Worker}_1!(\mathsf{d}_1{:}\mathsf{String}); \mathsf{Worker}_2!(\mathsf{d}_2{:}\mathsf{String}); \\ & \mathsf{Aggr}!(\mathsf{p}{:}\Sigma \mathsf{d}'{:}\mathsf{String}(\mathsf{d}), \mathsf{d}'_1{:}\mathsf{String}(\mathsf{d}_1), \mathsf{d}'_2{:}\mathsf{String}(\mathsf{d}_2).\mathsf{d}' = \mathsf{d}'_1 +\!\!+ \mathsf{d}'_2); \\ & \mathsf{Aggr}?(\mathsf{r}_3{:}\Sigma \mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1), \\ & \qquad \mathsf{r}_2{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2), \\ & \qquad \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2))); \\ & \mathsf{Client}!(\mathsf{res}{:}\Sigma \mathsf{d}'_1{:}\mathsf{String}(\mathsf{d}_1), \mathsf{d}'_2{:}\mathsf{String}(\mathsf{d}_2), \\ & \qquad \mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}'_1), \\ & \qquad \mathsf{r}_2{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}'_2), \\ & \qquad \mathsf{r}_3{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)). \\ & \qquad \mathsf{String}(\pi_1(\mathsf{r}_3))); \mathbf{end} \end{aligned}$$

The projection for the server role illustrates the key elements in our notion of endpoint projection. In the third message (the output to the aggregator), we bundle the information unknown by the aggregator in order to ensure the type is well-formed from the perspective of the recipient. Moreover, the usage of singletons preserves the dependencies specified in the global type (i.e. that the objects in question are indeed those received from the client and subsequently sent to the two worker endpoints). Note that in the input from the aggregator, the projected type does not require singletons since the server endpoint knows the identities of d_1 and d_2.

4 Value Dependent Processes and Typing

This section presents semantics and a typing system of value dependent processes.

$$
\begin{aligned}
G_{MR} \upharpoonright \mathsf{Client} \quad &\triangleq \mathsf{Server}!(\mathsf{d}{:}\mathsf{String});\\
&\quad \mathsf{Server}?(\mathsf{res}{:}\Sigma \mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String},\\
&\qquad \mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1),\\
&\qquad \mathsf{r}_2{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1),\\
&\qquad \mathsf{r}_3{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2)).\\
&\qquad\quad \mathsf{String}(\pi_1(\mathsf{r}_3)));\mathbf{end}\\
G_{MR} \upharpoonright \mathsf{Worker}_1 \quad &\triangleq \mathsf{Server}?(\mathsf{d}_1 : \mathsf{String});\\
&\quad \mathsf{Aggr}!(\mathsf{r}_1 : \Sigma \mathsf{r} : \mathsf{String}.r = f(d_1));\mathbf{end}\\
G_{MR} \upharpoonright \mathsf{Aggr} \quad &\triangleq \mathsf{Server}?(\mathsf{p}{:}\Sigma \mathsf{d}{:}\mathsf{String}, \mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String}.\mathsf{d} = \mathsf{d}_1 +\!\!+ \mathsf{d}_2);\\
&\quad \mathsf{Worker}_1?(\mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1));\\
&\quad \mathsf{Worker}_2?(\mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2));\\
&\quad \mathsf{Server}!(\mathsf{r}_3{:}\Sigma r{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2)));\mathbf{end}
\end{aligned}
$$

Fig. 6. Projections for G_{MR} – Client, Worker and Aggr roles.

4.1 Syntax and Operational Semantics

We define the process syntax, introducing the operational semantics, which is an extension of the synchronous multiparty session π-calculus studied in [11]. We use s to range over *session* names, c to range over *channels* which are either variables z, x or *session* names with *role* $s[\mathsf{p}]$, a to range over *shared* names. The process $\overline{a}[\mathsf{n}](z).P$ is a session initiation *request*, established through synchronisation by rule $\langle$Init$\rangle$, with the complementary accepting processes $a[\mathsf{p}](z).P$ (with $2 \leq \mathsf{p} \leq n$) on a shared channel a. We use $c[\mathsf{p}]$ in all session interactions, where c denotes a channel and p the participant implemented the by other endpoint process. Interactions within a session are: $c[\mathsf{p}]!(M); P$ sends the message M to participant p, continuing as P; and $c[\mathsf{p}]?(x); P$ receives a message or a channel from participant p, binding it to variable x in the continuation P (by rule $\langle$Com$\rangle$) where terms are reduced to values (denoted by $M \Downarrow V$); process $c[\mathsf{p}]!(s); P$ delegates channel s to participant p and continues as P (by rule $\langle$Del$\rangle$). $c[\mathsf{p}] \lhd l; P$ and $c[\mathsf{p}] \rhd l_i{:}P_i)_{i \in I}$ denote, respectively, selecting a label l by communicating with participant p and continuing as P and receiving a label l_i from participant p and continuing as P_i (by rule $\langle$Sel$\rangle$). Recursive process definitions $\mu X(x = M).P$ have the recursion variable x as a formal parameter, instantiated with M in the first iteration (we assume P always contains at least one recursive call on X) (by rule $\langle$Rec$\rangle$). The remaining operational semantics, structure congruence $\equiv$ and context rules which closed under parallel and shared and session name restrictions, are standard.

4.2 Typing System

We now introduce the typing system assigning local types to channels in processes. The key typing rules are given in Figure 8. We omit the rules that are not particular to our development, such as those for choice, branching, inactivity and session initiation for conciseness. We define the judgement $\Psi; \Gamma; \Delta \vdash P$, where Ψ is a typing context for message terms, Γ a mapping of shared names to global types and process variables to

$$\langle\text{Init}\rangle \quad \overline{a}[\mathsf{n}](z).P_1 \mid \textstyle\prod_{i\in\{2,\ldots,n\}} a[\mathsf{i}](z).P_i \to (\nu s)(\Pi_{i\in\{1,\ldots n\}} P_i\{s[\mathsf{i}]/z\}) \qquad s \notin fn(P_i)$$

$$\langle\text{Com}\rangle \quad s[\mathsf{p}][\mathsf{q}]!(M); P \mid s[\mathsf{q}][\mathsf{p}]?(x); Q \to P \mid Q\{V/x\} \qquad M \Downarrow V$$

$$\langle\text{Del}\rangle \quad s[\mathsf{p}][\mathsf{q}]!(s'[\mathsf{p}']); P \mid s[\mathsf{q}][\mathsf{p}]?(x); Q \to P \mid Q\{s'[\mathsf{p}']/x\}$$

$$\langle\text{Sel}\rangle \quad s[\mathsf{p}][\mathsf{q}] \triangleright \left(l_i{:}P_i\right)_{i\in I} \mid s[\mathsf{q}][\mathsf{p}] \triangleleft l_j; Q \to P_j \mid Q \qquad j \in I$$

$$\langle\text{Rec}\rangle \quad P\{M/x\}\{\mu X(x).P/X\} \mid R \to Q \implies \mu X(x = M).P \mid R \to Q$$

$$\langle\text{NuG}\rangle \quad P \to P' \implies (\nu a : \mathcal{G})P \to (\nu a{:}\mathcal{G})P'$$

$$\langle\text{NuS}\rangle \quad P \to P' \implies (\nu s)P \to (\nu s)P'$$

$$\langle\text{Par}\rangle \quad P \to P' \implies P \mid Q \to P' \mid Q$$

$$\langle\text{Cong}\rangle \quad (P \equiv P' \quad P' \to Q' \quad Q' \equiv Q) \implies P \to Q$$

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu a : \mathcal{G})P \mid Q \equiv (\nu a : \mathcal{G})(P \mid Q) \text{ if } a \notin fn(Q) \qquad (\nu s)P \mid Q \equiv (\nu s)(P \mid Q) \text{ if } s \notin fn(P)$$

$$(\nu a : \mathcal{G})(\nu a' : \mathcal{G}')P \equiv (\nu a' : \mathcal{G}')(\nu a : \mathcal{G})P \qquad (\nu s)(\nu s')P \equiv (\nu s')(\nu s)P$$

$$(\nu a : \mathcal{G})\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)\mathbf{0} \equiv \mathbf{0} \qquad (\nu a : \mathcal{G})(\nu s)P \equiv (\nu s)(\nu a : \mathcal{G})P$$

Fig. 7. Operational Semantics of Processes – Reduction and Structural Congruence

the specification of their variables, and Δ a (linear) mapping of channels to local types. The intuitive reading of the typing judgement is that P uses channels (and recursion variables) according to the types specified in Γ and Δ and message variables according to the types specified in Ψ. We write $\Gamma \vdash a{:}G$ iff $a{:}G \in \Gamma$. We also make use of a typing judgement for message terms $\Psi \vdash M : \tau$, denoting that M has type τ under the typing assumptions recorded in Ψ. We omit this typing judgement for the sake of generality of the underlying type theory. We recall the requirement of the usual type safety results of progress and type preservation (and so, a substitution principle) in the presence of singleton types and subtyping (as detailed in Section 3.1).

Rule (VSEND) types sending of data messages. Sending a datum M binds it to x in the continuation type, as expected in a (value) dependently-typed setting. Sending a channel requires its existence in the context with the appropriate type. Dually, rule (VRECV) types the reception of data, where the process that expects to receive a data message of type τ is warranted to use it in its continuation.

The typing rule for (REC) recursive process definitions assigns a channel with a parameterised recursive type by registering in Γ the necessary information regarding the process recursion variable (the channel name, the recursive type variable and the parameter variable), where the local typing environment must be empty. Rule (VAR) simply matches the process recursion variable with the type recursion variable according to the information if Γ, checking that the recursive parameter is appropriately typed. The remaining rules are standard in the MPST literature [9,22,11], of which we highlight the

$$\text{(VSEND)}\ \frac{\Psi \vdash M{:}\tau \quad \Psi; \Gamma; \Delta, c{:}T\{M/x\} \vdash P}{\Psi; \Gamma; \Delta, c{:}\mathsf{p}!(x{:}\tau).T \vdash c[\mathsf{p}]!(M); P} \qquad \text{(VRECV)}\ \frac{\Psi, x{:}\tau; \Gamma; \Delta, c{:}T \vdash P}{\Psi; \Gamma; \Delta, c{:}\mathsf{p}?(x{:}\tau).T \vdash c[\mathsf{p}]?(x); P}$$

$$\text{(REC)}\ \frac{\Psi \vdash M : \tau \quad \Psi, x{:}\tau; \Gamma, X : (c, t, x); c{:}T \vdash P}{\Psi; \Gamma; c{:}\mu t(x = M : \tau).T \vdash \mu X(x = M).P}$$

$$\text{(VAR)}\ \frac{\Psi, x{:}\tau \vdash M : \tau}{\Psi, x{:}\tau; \Gamma, X{:}(c, t, x); c{:}t\langle M\rangle \vdash X\langle M\rangle} \qquad \text{(SUB)}\ \frac{\Psi \vdash T \leq T' \quad \Psi; \Gamma; \Delta, c{:}T \vdash P}{\Psi; \Gamma; \Delta, c{:}T' \vdash P}$$

$$\text{(SRES)}\ \frac{\Psi; \Gamma; \Delta, s[1]{:}T_1, \ldots s[n] : T_n \vdash P \quad \mathsf{co}(s[1]{:}T_1, \ldots s[n] : T_n)}{\Psi; \Gamma; \Delta \vdash (\boldsymbol{\nu} s)P}$$

Fig. 8. Local Typing Rules.

(SRES) rule for session channel restriction, requiring that all the several role annotatted endpoints be *coherent* (Definition 4.5). Coherence relies on a notion of partial projection (Definition 4.2) and session duality (Definition 4.3), which we now introduce.

Partial projection is defined as a function taking an endpoint type and a role identifier. Intuitively, it extracts from the endpoint type the behaviour that pertains only to the specified role, erasing all role annotations (since the type is now completely localised to a single role – i.e. a binary session type). We range over binary session types with S, T. Duality is defined in the natural way, matching inputs with outputs (appealing to subtyping in the case for data communication to ensure compatibility of the data types); and branching with selection.

Definition 4.1 (Binary Type Merge). *Let* $T = \oplus\left(\mathsf{l}_i : T_i\right)_{i \in I}$ *and* $T' = \oplus\left(\mathsf{l}'_j : T'_j\right)_{j \in J}$. *The merge* $T \sqcup T'$ *of* T *and* T' *is defined as:*

$$T \sqcup T' \triangleq \oplus\left(\mathsf{l}_h : T_h\right)_{h \in I \setminus J} \cup \left(\mathsf{l}'_h : T'_h\right)_{h \in J \setminus I} \cup \left(\mathsf{l}_h : T_h \sqcup T'_h\right)_{h \in I \cap J}$$
$$T \sqcup T \triangleq T$$

if $\mathsf{l}_h = \mathsf{l}'_h$ *for each* $h \in I \cap J$. *Merge is homomorphic (i.e.* $\mathcal{C}[T_1] \sqcup \mathcal{C}[T_2] = \mathcal{C}[T_1 \sqcup T_2]$*) and is undefined otherwise.*

Definition 4.2 (Partial Projection). *The partial projection of a local type* T *onto* p*, denoted by* $T \upharpoonright \mathsf{p}$*, is defined by the rules of Figure 9.*

Definition 4.3 (Duality). *The duality relation between projections of local types is the minimal symmetric relation satisfying:*

$$\mathbf{end} \bowtie \mathbf{end} \qquad t\langle M\rangle \bowtie t\langle M'\rangle$$
$$T \bowtie T' \Longrightarrow (\mu t(x = M : \tau).T) \bowtie (\mu t(x = M' : \tau).T')$$
$$T \bowtie T' \wedge \tau \leq \tau' \Longrightarrow !(x{:}\tau); T \bowtie ?(x{:}\tau'); T'$$
$$T \bowtie T' \Longrightarrow !(U); T \bowtie ?(U); T'$$
$$\forall i \in I\ T_i \bowtie T'_i \Longrightarrow \oplus(l_i{:}T_i) \bowtie \&(l_i : T'_i)$$

$$
\begin{aligned}
(\mathsf{r}!(x{:}\tau);T)\upharpoonright \mathsf{p} &= \begin{cases} !(x:\tau);(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright \mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}?(x{:}\tau);T)\upharpoonright \mathsf{p} &= \begin{cases} ?(x{:}\tau);(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright \mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}!(U);T)\upharpoonright \mathsf{p} &= \begin{cases} !(U);(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright \mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}?(U);T)\upharpoonright \mathsf{p} &= \begin{cases} ?(U);(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright \mathsf{p} & \text{otherwise}\end{cases}\\
(\oplus\mathsf{r}(l_i{:}T_i)_{i\in I})\upharpoonright \mathsf{p} &= \begin{cases} \oplus(l_i{:}(T_i\upharpoonright \mathsf{p}))_{i\in I} & \text{if } \mathsf{p}=\mathsf{r}\\ \sqcup_{i\in I}(T_i\upharpoonright \mathsf{p}) & \text{otherwise}\end{cases}\\
(\&\mathsf{r}(l_i{:}T_i)_{i\in I})\upharpoonright \mathsf{p} &= \begin{cases} \&(l_i{:}(T_i\upharpoonright \mathsf{p}))_{i\in I} & \text{if } \mathsf{p}=\mathsf{r}\\ \sqcup_{i\in I}(T_i\upharpoonright \mathsf{p}) & \text{otherwise}\end{cases}\\
(\mu t(x=M:\tau).T)\upharpoonright \mathsf{p} &= \begin{cases} \mu t(x=M:\tau).(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}\in\mathcal{T} \text{ and } \mathsf{p} \text{ knows } M\\ \mu t.(T\upharpoonright \mathsf{p}) & \text{if } \mathsf{p}\in T \text{ and } \mathsf{p} \text{ doesn't know } M\\ \mathbf{end} & \text{otherwise}\end{cases}\\
t\langle M\rangle\upharpoonright \mathsf{p} &= \begin{cases} t\langle M\rangle & \text{if } \mathsf{p} \text{ knows } M\\ t & \text{otherwise}\end{cases}\\
\mathbf{end}\upharpoonright \mathsf{p} &= \mathbf{end}
\end{aligned}
$$

Fig. 9. Partial Projection.

Duality is crucial to ensure compatibility between the different participants in a multiparty conversions. This notion is made precise in the following lemma.

Lemma 4.4. *Let G be a global type and $\mathsf{p}\neq\mathsf{q}$. Then $(G\upharpoonright\mathsf{p})\upharpoonright\mathsf{q}\bowtie(G\upharpoonright\mathsf{q})\upharpoonright\mathsf{p}$.*

Coherence thus ensures compatibility of all participants in a multiparty session, requiring that for each role in the multiparty conversation, all performed actions are matched by a dual action performed by the expected recipient.

Definition 4.5 (Coherence). *A session environment Δ is coherent for the session s, written $\mathsf{co}(\Delta,s)$ if $s[\mathsf{p}]:T\in\Delta$ and $s[\mathsf{q}]:T'\in\Delta$ imply $T\upharpoonright\mathsf{q}\bowtie T'\upharpoonright\mathsf{p}$. A session environment is coherent if it is coherent for all sessions which occur in it.*

5 Safety Properties of Value Dependencies

This section lists the main properties of the typing system. Recalling the notion of *history sensitivity* (Definition 2.2), which intuitively requires that in a global type G, for each interaction in which s sends some data of type τ, all free variables of type τ must have been defined in a previous interaction of G involving s, we state soundness of compatible type binding generation.

Theorem 5.1 (Compatible type binding generation is sound). *Let G be a well-formed, history sensitive global type such that $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G' \sqsubseteq G$. We have that the pair of data types $((CTB(x{:}\tau), (CTB(x{:}\tau))^{\dagger})$ is compatible with p and q.*

Intuitively, it is easy to see that CTB generates pairs of compatible types given the subtyping rules for singletons (and Definition 3.5 of singleton erasure), combined with the fact that CTB collects only data known by p and unknown by q, relevant to the message exchange of type τ.

Given that types intrinsically specify properties of exchanged data, subject congruence and reduction ensure that any well-typed process is guaranteed to conform with its behavioural specification in a strong sense. Subject reduction relies crucially on a substitution principle for types and processes, which is a lifted version of the substitution principle for the dependent data layer.

Lemma 5.2 (Term Substitution). *If $\Psi, x{:}\tau; \Gamma; \Delta \vdash P$ and $\Psi \vdash M{:}\tau$ then we have that $\Psi; \Gamma; \Delta\{M/x\} \vdash P\{M/x\}$.*

Theorem 5.3 (Subject Congruence and Reduction). *If $\Psi; \Gamma; \Delta \vdash P$ and $P \equiv Q$ then there is $\Delta' \equiv \Delta$ such that $\Psi; \Gamma; \Delta' \vdash Q$; and If $\Gamma \vdash P$ and $P \to P'$ then $\Gamma \vdash P'$.*

Note that adherence to the properties of data specified in types is intrinsic: values occurring in well-typed processes act as proof witnesses to the stated properties which are thus inherently satisfied, entailing a notion of communication safety, see [9, Th. 5.5].

Error Freedom In order to characterise the kind of communication errors that are disallowed by our typing discipline, we define a notion of *extended* process which explicitly references message typing information by considering the following process constructs for (data) input and output:

$$P, Q ::= c[\mathsf{p}]!(M)\{x{:}\tau\}; P \mid c[\mathsf{p}](x)\{x{:}\tau\}; P \mid \dots$$

We then define a typed reduction and labelled semantics, written $\longmapsto$ and $\overset{\alpha}{\longmapsto}$, respectively. Typed reduction $\longmapsto$ is defined by the same rules as those of Figure 7 but where the message synchronisation rule is replaced with (error is a special process construct denoting the error state):

$$\frac{M \Downarrow V \quad \cdot \vdash \tau \leq \tau' \quad \cdot \vdash V : \tau}{s[\mathsf{p}][\mathsf{q}]!(M)\{x{:}\tau\}; P \mid s[\mathsf{q}][\mathsf{p}]?(x)\{x{:}\tau'\}; Q \longmapsto P\{V/x\} \mid Q\{V/x\}}$$

$$\frac{M \Downarrow V \quad \cdot \not\vdash \tau \leq \tau' \vee \cdot \not\vdash V : \tau}{s[\mathsf{p}][\mathsf{q}]!(M)\{x{:}\tau\}; P \mid s[\mathsf{q}][\mathsf{p}]?(x)\{x{:}\tau'\}; Q \longmapsto \mathsf{error}}$$

Equipped with extended processes and typed reduction, we may then show that well-typed (extended) processes never reach an error state.

Theorem 5.4 (Error Freedom). *Let $\Gamma \vdash P$ and $P \longmapsto^{*} P'$. Then error is not a subterm of P'.*

6 Specification without Communication

So far we have mostly been concerned with the challenges of certifying data exchanges in a multiparty setting by having process endpoints exchange explicit proof objects. However, it is quite often the case that we may wish to reference data and constraints that are reasonable at a specification level but that have little computational interest at runtime. For instance, consider the following global type,

$$\mathsf{p} \to \mathsf{q} : (x{:}\mathsf{Nat}).\mathsf{p} \to \mathsf{q} : (y{:}x > 2).G \tag{6}$$

In the example above, participant p sends q some natural number x, followed by a proof denoting that x is greater than 2. While such an exchange does ensure that a well-typed implementation of the endpoint p must necessarily send to q an integer greater than 2, the endpoint q may have little interest in actually receiving a proof that $x > 2$. Rather, the exchange denoted by the second message from p to q appears as an encoding artefact due to the fact that the framework requires explicit proof exchanges by default.

While it is the case that we could omit a second exchange by "currying" the two communication actions into a pair,

$$\mathsf{p} \to \mathsf{q} : (x{:}\Sigma a{:}\mathsf{Nat}.a > 2).G \tag{7}$$

the issue still remains that we are forced to send potentially unnecessary data.

We can alleviate this issue through the usage of a proof irrelevance modality, written $[\tau]$, denoting that there exists a term of type τ (and thus, a proof of τ), but the identity of the term itself is deemed computationally irrelevant. To make this notion precise, we appeal to a new class of typing assumptions $x \div \tau$, meaning that x stands for a term of type τ that is not computationally available; and to a promotion operation on contexts (written $\Psi^{\oplus}$) mapping computationally irrelevant assumptions to ordinary ones:

$$\frac{\Psi^{\oplus} \vdash M : \tau}{\Psi \vdash [M] : [\tau]}\ ([]\mathrm{I}) \qquad \frac{\Psi \vdash M : [\tau] \quad \Psi, x \div \tau \vdash N : \sigma}{\Psi \vdash \mathsf{let}\ [x] = M\ \mathsf{in}\ N : \sigma}\ ([]\mathrm{E})$$

Given that we are only warranted in using irrelevant assumptions within proof irrelevant terms, it is easy to see that proof irrelevance cannot affect the computational outcome of a program and so we may consistently erase proof irrelevant terms at runtime.

We combine this notion of proof irrelevance with an erasure operation that eliminates communication of proof irrelevant terms – since they may not be used in a computationally significant way, they bear no impact on the computational outcome of the session. For instance, we may rewrite the global type of (7) as:

$$\mathsf{p} \to \mathsf{q} : (x{:}\Sigma a{:}\mathsf{Nat}.[a > 2]).G \tag{8}$$

marking that the proof of $a > 2$ is not computationally significant, but must exist during type-checking.

With the combined use of proof irrelevance and erasure we ensure that the specification must still hold, in the sense that the proof objects must be present in endpoint processes for the purposes of type-checking, but are then omitted at runtime to minimise potentially unnecessary communication. An alternative approach, only feasible for decidable theories, would be to generate proofs automatically by appealing to some external decision procedure.

6.1 Erasure of proof irrelevant terms

We introduce a simple erasure procedure on types, processes and terms that replaces proof irrelevance with the unit type (and the unit element at the term level). Recall that since proof irrelevant terms have no bearing on the computational outcome of programs, the erasure is safe w.r.t the behaviour of programs.

Definition 6.1 (Erasure). *We inductively define erasure on local types, terms and processes, written $T^{\downarrow}$ (resp. $\tau^{\downarrow}$, $M^{\downarrow}$ and $P^{\downarrow}$) by the following rules (we show only the most significant cases, all others simply traverse the underlying structure inductively):*

$$
\begin{array}{llll}
(\mathsf{p}!(x{:}\tau);T)^{\downarrow} & \triangleq \mathsf{p}!(x{:}\tau^{\downarrow});T^{\downarrow} & (\mathsf{p}!(x{:}\tau);T)^{\downarrow} & \triangleq \mathsf{p}?(x{:}\tau^{\downarrow});T^{\downarrow} \\
(\oplus\mathsf{p}(\mathsf{l}_p{:}T_i)_{i\in I})^{\downarrow} & \triangleq \oplus\mathsf{p}(\mathsf{l}_p{:}T_i\downarrow)_{i\in I}) & (\&\mathsf{p}(\mathsf{l}_p{:}T_i)_{i\in I})^{\downarrow} & \triangleq \&\mathsf{p}(\mathsf{l}_p{:}T_i^{\downarrow})_{i\in I} \\
(\mathsf{p}!(U);T)^{\downarrow} & \triangleq \mathsf{p}!(U^{\downarrow});T^{\downarrow} & (\mathsf{p}?(U);T)^{\downarrow} & \triangleq \mathsf{p}?(U^{\downarrow});T^{\downarrow} \\
(\mu t.(x=M:\tau).T)^{\downarrow} & \triangleq \mu t.(x=M^{\downarrow}:\tau^{\downarrow}).T^{\downarrow} & t\langle M\rangle^{\downarrow} & \triangleq t\langle M^{\downarrow}\rangle
\end{array}
$$

$$
\begin{array}{llll}
(\overline{a}[\mathsf{n}](z).P)^{\downarrow} & \triangleq \overline{a}[\mathsf{n}](z).P^{\downarrow} & (a[\mathsf{p}](z).P)^{\downarrow} & \triangleq a[\mathsf{p}](z).P^{\downarrow} \\
(c[\mathsf{p}]!(M);P)^{\downarrow} & \triangleq c[\mathsf{p}]!(M^{\downarrow});P^{\downarrow} & (c[\mathsf{p}]!(s);P)^{\downarrow} & \triangleq c[\mathsf{p}]!(s);P^{\downarrow} \\
(c[\mathsf{p}]?(x);P)^{\downarrow} & \triangleq c[\mathsf{p}]?(x);P^{\downarrow} & (c[\mathsf{p}]\lhd l;P)^{\downarrow} & \triangleq c[\mathsf{p}]\lhd l;P^{\downarrow} \\
(c[\mathsf{p}]\rhd \left(l_i{:}P_i\right)_{i\in I})^{\downarrow} & \triangleq c[\mathsf{p}]\rhd \left(l_i{:}P_i^{\downarrow}\right)_{i\in I} & (\mu X(x=M).P)^{\downarrow} & \triangleq \mu X(x=M^{\downarrow}).P^{\downarrow} \\
X\langle M\rangle^{\downarrow} & \triangleq X\langle M^{\downarrow}\rangle & &
\end{array}
$$

$$
\begin{array}{llll}
(\Pi x:\tau.\sigma)^{\downarrow} & \triangleq \Pi x:\tau^{\downarrow}.\sigma^{\downarrow} & (\Sigma x:\tau.\sigma)^{\downarrow} & \triangleq \Sigma x:\tau^{\downarrow}.\sigma^{\downarrow} \\
b^{\downarrow} & \triangleq b & \mathcal{S}(M)^{\downarrow} & \triangleq \mathcal{S}(M^{\downarrow}) \\
[\tau]^{\downarrow} & \triangleq \mathsf{unit} & (\Diamond_{\mathsf{p}}\tau)^{\downarrow} & \triangleq \Diamond_{\mathsf{p}}\tau^{\downarrow}
\end{array}
$$

$$
\begin{array}{llll}
[M]^{\downarrow} & \triangleq \langle\rangle & (\lambda x.M)^{\downarrow} & \triangleq \lambda x.M^{\downarrow} \\
\langle M,N\rangle^{\downarrow} & \triangleq \langle M^{\downarrow},N^{\downarrow}\rangle & \langle\rangle^{\downarrow} & \triangleq \langle\rangle
\end{array}
$$

The goal of the erasure function above is to essentially replace all instances of proof irrelevant objects with the unit element $\langle\rangle$. We note that it is not in general the case that $(G \upharpoonright \mathsf{p})^{\downarrow} = (G^{\downarrow}) \upharpoonright \mathsf{p}$, since projection of an erased global type may not need to preserve some dependencies that were present in the original global type. We may then consistently erase communication of messages of unit type.

Definition 6.2 (Communication Erasure). *We write T^*, P^*, M^* and τ^* for the following erasure (the remaining cases are obtained by homomorphic extension):*

$$(\mathsf{p}\to\mathsf{q}:(x{:}\mathsf{unit}).G)^* \triangleq G^* \qquad (\mathsf{p}!(x{:}\mathsf{unit}).T)^* \triangleq T^* \qquad (\mathsf{p}?(x{:}\mathsf{unit}).T)^* \triangleq T^*$$

To summarise, our proposed methodology for the usage of ghost variables in specifications is to mark specification-level terms as proof irrelevant at the level of global types. Projection is then performed on the global type, propagating the proof irrelevance accordingly to the local types which we use to type the several endpoint processes (which contain the explicit proof objects, now marked as proof irrelevant).

Having successfully checked the endpoints, we may then perform the erasure procedure(s) described above: by performing the erasure $\downarrow$ on the original global type, we generate local types in which instances of proof irrelevance have been replaced with unit; by applying the erasure * we eliminate irrelevant communication actions from

types, erasing those actions from the endpoints and refactoring message payloads accordingly. We thus remove unnecessary communication actions but ensure that specified properties are satisfied.

MapReduce In our running example of a certified MapReduce-style computation, where the global type G_{MR} specifies that the workers and aggregator endpoints indeed perform the appropriate computation, we may easily apply the ghost variable technique introduced in this section to eliminate several potentially unnecessary communication steps, including the potentially excessive passing of data that appears in local types as an artefact of endpoint projection.

Consider the following revision of G_{MR}, referred to as $G_{[MR]}$, where we mark the message from the server to the aggregator which informs of the partition of data as proof irrelevant ($\mathsf{Server} \to \mathsf{Aggr} : (\mathsf{p} : [\mathsf{d} = \mathsf{d}_1 +\!\!+ \mathsf{d}_2])$), and similarly for the proofs that the sent data objects are indeed the results of performing the specified computation ($\mathsf{Worker}_i \to \mathsf{Aggr} : (\mathsf{r}_i : \Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_i)]$ and $\mathsf{Aggr} \to \mathsf{Server} : (\mathsf{r}_3 : \Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2))])$). We then have the following endpoint projections:

$$
\begin{aligned}
G_{[MR]} \upharpoonright \mathsf{Client} \triangleq\ & \mathsf{Server}!(\mathsf{d}{:}\mathsf{String});\\
& \mathsf{Server}?(\mathsf{res}{:}\Sigma \mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String},\\
& \qquad \mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_1)],\\
& \qquad \mathsf{r}_2{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_1)],\\
& \qquad \mathsf{r}_3{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)]).\\
& \qquad\ \mathsf{String}(\pi_1(\mathsf{r}_3))); \mathbf{end}\\
G^{\downarrow}_{[MR]} \upharpoonright \mathsf{Client} \triangleq\ & \mathsf{Server}!(\mathsf{d}{:}\mathsf{String});\\
& \mathsf{Server}?(\mathsf{res}{:}\Sigma \mathsf{r}_3{:}(\Sigma \mathsf{r}{:}\mathsf{String.unit}).\mathsf{String}(\pi_1(\mathsf{r}_3)))
\end{aligned}
$$

By performing the erasure before projecting, we eliminate a substantial amount of dependency information from the local type for the Client. We note that transforming processes satisfying $G_{[MR]} \upharpoonright \mathsf{Client}$ into ones satisfying $G^{\downarrow}_{[MR]} \upharpoonright \mathsf{Client}$ can easily be achieved by some simple program transformations, related to those used in program generation for dependently typed functional languages [18,14].

6.2 Soundness of Erasure

We make precise the static and dynamic soundness of our erasure procedures. The static safety theorem (Theorem 6.3) states that a consistent usage of erasures does not violate the typing discipline.

Theorem 6.3 (Static Safety). *Let $\Psi; \Gamma; \Delta \vdash P$. Then we have that:*

(a) $\Psi^{\downarrow}; \Gamma^{\downarrow}; \Delta^{\downarrow} \vdash P^{\downarrow}$
(b) $(\Psi^{\downarrow})^*; (\Gamma^{\downarrow})^*; (\Delta^{\downarrow})^* \vdash (P^{\downarrow})^*$

We also show that erased processes have an operational correspondence with their unerased counterparts.

Theorem 6.4 (Operational Correspondence). *Let $\Gamma \vdash P$:*

(a) If $P \to P'$ *then* $P^{\downarrow} \to P'^{\downarrow}$
(b) If $(P^{\downarrow})^* \to P'$ *then* $P \to^* Q$ *such that* $(Q^{\downarrow})^* \equiv P'$.

For the erasure of Definition 6.1, we have a very precise operational correspondence by virtue of the computational insignificance of proof irrelevant terms (Theorem 6.4 (a)). For the communication erasure of Definition 6.2, the processes in the image of the erasures may naturally produce less reductions, which account for the erased communication steps. However, we can ensure that a reduction in an erased process $(P^{\downarrow})^* \to P'$ can be matched by potentially a sequence of reductions in the original process P, such that applying the two erasures to the reduct of P produces a structurally equivalent process to P' (Theorem 6.4, (b)).

7 Conclusion

In this section we present some additional discussion of key design choices and avenues of future work and conclude.

7.1 Further Discussion

We discuss two fundamental considerations: some of the particulars of compatible type binding generation and how they may be potentially simplified through a fundamental use of proof irrelevance akin to that of Section 6; and how our design choices affect a potential implementation of the language discussed in this paper, specifically what should be verified statically and/or dynamically and in what circumstances does the type system help guide these choices.

Compatible Type Binding Generation In Section 3 we have discussed the challenges of defining projection in the presence of value dependencies, given that each participant only has a partial view of the global protocol and thus the notion of projection from previous works on multiparty sessions produce endpoint types that are either not well-formed or fail to capture the appropriate data restrictions specified in global types.

We address this issue by bundling in each message exchange all the information that is unknown by message recipients such that the endpoint types for senders and receivers are well-formed, compatible and preserve the intended semantics of global types. A potential disadvantage of our approach is that the bundling procedure can insert a non-trivial amount of extra information in message exchanges. For instance, in the global type:

$$\begin{aligned} G \triangleq\ & A \to B : (x_1{:}\tau_1). \\ & A \to B : (x_2{:}\tau_2). \\ & \quad \vdots \\ & A \to B : (x_n{:}\tau_n). \\ & A \to C : (y{:}\Sigma z : \tau.\mathcal{P}(z, x_1, \ldots, x_n)) \end{aligned}$$

Participant A sends to participant B a sequence of n messages, after which it sends to participant C a message of type $\Sigma z : \tau.\mathcal{P}(z, x_1, \ldots, x_n)$, such that C does not know

x_1 through x_n. In this scenario, when projecting the exchange of message y, A must not only send the object of type $\Sigma z : \tau.\mathcal{P}(x_1, \ldots, x_n)$ but also the n messages previously sent to participant B which are unknown to C.

In Section 6 we have introduced a way of minimising potentially unnecessary communications through the usage of proof irrelevance and type-oriented erasure, where certain portions of data types are marked as irrelevant and subsequently erased. At the level of compatible type binding generation, it should also be possible to directly make use of proof irrelevance to reduce the communication overheads mentioned above, given that unknown message variables have a somewhat proof irrelevant flavour (i.e. they cannot be used precisely because they are unknown).

One potential approach to this issue is to introduce a proof-irrelevant Σ-type, written $\Sigma x \div \tau.\sigma$ and modify compatible type binding generation to quantify over unknown variables using these proof irrelevant pairs. The main issue is that it forces type families using such proof irrelevant variables to be themselves proof irrelevant. We leave these challenges for future work.

On Implementation The main contribution of this paper is the conceptual extension of the multiparty session typed framework with value dependencies, the language presented in this paper leads itself towards more realistic implementation considerations. Our basic foundation is that proofs are exchanged as witnesses to the properties specified in types. These proof objects are no more than terms from a dependently-typed language, but one may wonder if it is indeed feasible to explicitly exchange proof objects instead of somehow verifying the necessary properties dynamically. Regardless, given the potential distributed nature of the framework, it seems natural to require that communicated proof objects be checked at runtime on the recipient side.

While there are circumstances where proof objects may not be exchanged and instead generated (or have the necessary properties checked) at runtime on the receiver side, this requires the property language to be decidable, which may be too restrictive (we note that we require proof *checking* to be decidable, which is a significantly weaker condition), although a potentially reasonable assumption in certain application scenarios, or for settings where programmers cannot be expected to write proof objects by hand. Another alternative worth considering is to have participants exchange digitally signed objects that hold them accountable for the existance of certain proofs.

While there seems to be a somewhat flexible range of possibilities in terms of proof communication, it is unavoidable that an implementation of the language integrates both static and dynamic checking in order to ensure that the specified properties indeed hold throughout execution. To this end, an extension of the system where trust amongst session participants is determined *a priori* might help guide the static/dynamic verification procedures. For instance, among trusted participants one may avoid dynamic checks entirely, among "somewhat" trusted participants, digital proof certificates might be required, but not the complete proof objects, whereas communication with untrusted agents would require the full range of dynamic and static checks.

We are currently investigating some of these avenues of research, in particular how the exchange of signed certificates interacts with assurances of data provenance. Another aspect that needs to be studied is the potential leakage of sensitive information that may occur while generating compatible endpoint types due to unknown message

dependencies, which should be alleviated by the techniques discussed in the sections above.

7.2 Concluding Remarks

We shall conclude this paper with a quotation from the end of Ambition and Vision in [1] to record what we promised for the future:

Ambition and Vision [1]. *As with data types, we expect session types to play a role in all aspects of software. Today,* architects *model systems using types that are directly supported in the programming language, whereas they model communications using protocols that have no direct support in the programming language; tomorrow, they will model communication using session types that are directly supported in the programming language. Today,* programmers *use interactive development environments that prompt for methods based on types, and give immediate feedback indicating where code violates type discipline, whereas they have no similar support for coding communications; tomorrow, interactive development environments will prompt for messages based on session types, and give immediate feedback indicating where code violates session type discipline. Today,* software tools *exploit types to optimise code, whereas they do not exploit protocols; tomorrow, software tools will exploit session types to optimise communication. In short, architects, programmers, and software tools will all be aided by session types to reduce the cost of producing concurrent and distributed software, while increasing its reliability and efficiency.*

The work introduced in this paper is a small step to move towards the ambition and vision of the ABCD project statement, proposing a session type discipline that builds on the current usages of data types as a tool for modelling, developing and improving modern software, integrating them at a deep level with MPST in order to provide a unified framework for statically certified programs, from computation to communication.

References

1. ABCD. A basis for concurrency and distribution. http://groups.inf.ed.ac.uk/abcd/.
2. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *21st Int. Conf. Concur. Theory*, pages 162–176. LNCS 6269, 2010.
3. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, pages 222–236, 2010.
4. L. Caires, F. Pfenning, and B. Toninho. Towards concurrent type theory. In *Types in Language Design and Implementation*, pages 1–12, 2012.
5. M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A gentle introduction to multiparty asynchronous session types. In *SFM'15*, pages 146–178, 2015.
6. P. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.
7. S. Gay and M. Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
8. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, pages 122–138, 1998.

9. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.
10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
11. D. Kouzapas and N. Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10(4), 2014.
12. S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *ESOP'15*, pages 560–584, 2015.
13. MRG. Mobility reading group. http://mrg.doc.ic.ac.uk/.
14. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
15. F. Pfenning, L. Caires, and B. Toninho. Proof-carrying code in a session-typed process calculus. In *CPP*, pages 21–36, 2011.
16. C. A. Stone and R. Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722, 2006.
17. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, pages 398–413, 1994.
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4pl2*, 2013.
19. B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11*, pages 161–172, 2011.
20. P. Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286, 2012.
21. P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.
22. N. Yoshida, P. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCS'10*, pages 128–145, 2010.