Statically Verified Refinements for Multiparty Protocols

FANGYI ZHOU, Imperial College London, United Kingdom
FRANCISCO FERREIRA, Imperial College London, United Kingdom
RAYMOND HU, University of Hertfordshire, United Kingdom
RUMYANA NEYKOVA, Brunel University London, United Kingdom
NOBUKO YOSHIDA, Imperial College London, United Kingdom

With distributed computing becoming ubiquitous in the modern era, safe distributed programming is an open challenge. To address this, multiparty session types (MPST) provide a typing discipline for message-passing concurrency, guaranteeing communication safety properties such as deadlock freedom.

While originally MPST focus on the communication aspects, and employ a simple typing system for communication payloads, communication protocols in the real world usually contain *constraints* on the payload. We introduce *refined multiparty session types (RMPST)*, an extension of MPST, that express data dependent protocols via *refinement types* on the data types.

We provide an implementation of RMPST, in a toolchain called SESSION⋆, using SCRIBBLE, a toolchain for multiparty protocols, and targeting F⋆, a verification-oriented functional programming language. Users can describe a protocol in SCRIBBLE and implement the endpoints in F⋆ using *refinement-typed* APIs generated from the protocol. The F⋆ compiler can then statically verify the refinements. Moreover, we use a novel approach of callback-styled API generation, providing *static* linearity guarantees with the inversion of control. We evaluate our approach with real world examples and show that it has little overhead compared to a naïve implementation, while guaranteeing safety properties from the underlying theory.

148

CCS Concepts: • **Theory of computation** → **Automated reasoning**; *Distributed computing models*; • **Software and its engineering** → **Source code generation**.

Additional Key Words and Phrases: Multiparty Session Types (MPST), Refinement Types, Code Generation, F⋆, Distributed Programming

1 INTRODUCTION

Distributed interactions and message passing are fundamental elements of the modern computing landscape. Unfortunately, language features and support for high-level and *safe* programming of communication-oriented and distributed programs are much lacking, in comparison to those enjoyed for more traditional "localised" models of computation. One of the research directions towards addressing this challenge is *concurrent behavioural types* [Ancona et al. 2016; Gay and

Authors' addresses: Fangyi Zhou, Imperial College London, United Kingdom, fangyi.zhou15@imperial.ac.uk; Francisco Ferreira, Imperial College London, United Kingdom, f.ferreira-ruiz@imperial.ac.uk; Raymond Hu, University of Hertfordshire, United Kingdom, r.z.h.hu@herts.ac.uk; Rumyana Neykova, Brunel University London, United Kingdom, rumyana.neykova@ brunel.ac.uk; Nobuko Yoshida, Imperial College London, United Kingdom, n.yoshida@imperial.ac.uk.

$$G = \mathbf{A} \rightarrow \mathbf{B} : Count(count : \mathtt{int}\{count \geq 0\}).$$
$$\mu\mathtt{t}(curr : \mathtt{int}\{curr \geq 0 \wedge curr \leq count\})\langle curr := 0\rangle.$$
$$\mathbf{B} \rightarrow \mathbf{C} \begin{cases} Hello(it : \mathtt{int}\{curr < count \wedge it = count\}).\mathtt{t}\langle curr := curr + 1\rangle \\ Finish(it : \mathtt{int}\{curr = count \wedge it = count\}).\mathtt{end} \end{cases}$$

A Global Type G

Projection onto each Participant

Local Type for $\mathbf{A}$　$L_{\mathbf{A}}$　　　Local Type for $\mathbf{B}$　$L_{\mathbf{B}}$　　　Local Type for $\mathbf{C}$　$L_{\mathbf{C}}$
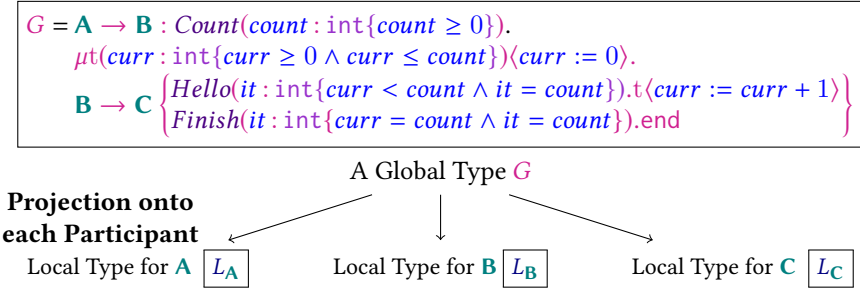
Fig. 1. Top-down View of (R)MPST

Ravara 2017], which seek to extend the benefits of conventional type systems, as the most successful form of lightweight formal verification, to communication and concurrency.

Multiparty session types (MPST) [Honda et al. 2008, 2016], one of the most active topics in this area, offer a theoretical framework for specifying message passing protocols between multiple participants. MPST use a type system–based approach to *statically* verify whether a system of processes implements a given protocol safely. The type system guarantees key execution properties such as freedom from message reception errors or deadlocks. However, despite much recent progress, there remain large gaps between the current state of the art and (i) powerful and *practical* languages and techniques available to programmers today, and (ii) more advanced type disciplines needed to express a wider variety of constraints of interaction found in real-world protocols.

This paper presents and combines two main developments: a theory of MPST enriched with *refinement types* [Freeman and Pfenning 1991], and a practical method, *callback-based programming*, for safe session programming. The key ideas are as follows:

Refined *Multiparty Session Types (RMPST)*. The MPST theory [Honda et al. 2008, 2016] provides a core framework for decomposing (or *projecting*) a *global type* structure, describing the collective behaviour of a distributed system, into a set of participant-specific *local types* (see Fig. 1). The local types are then used to implement endpoints.

Our theory of RMPST follows the same top-down methodology, but enriches MPST with features from *refinement types* [Freeman and Pfenning 1991], to support the elaboration of data types in global and local types. Refinement types allow *refinements* in the form of logical predicates and constraints to be attached to a base type. This allows to express various constraints in protocols.

To motivate our refinement type extension, we use a counting protocol shown in Fig. 1, and leave the details to § 4. Participant $\mathbf{A}$ sends $\mathbf{B}$ a number with a *Count* message. In this message, the refinement type *count* : $\mathtt{int}\{count \geq 0\}$ restricts the value for *count* to be a natural number. Then $\mathbf{B}$ sends $\mathbf{C}$ exactly that number of *Hello* messages, followed by a *Finish* message.

We demonstrate how refinement types are used to *better* specify the protocol: The counting part of the protocol is described using a recursive type with two branches, where we use refinement types to restrict the protocol flow. The variable *curr* is a *recursion variable*, which remembers the current iteration, initialised to 0, and increments on each recursion (*curr* := *curr* + 1). The refinement *curr* = *count* in the *Finish* branch specifies that the branch may only be taken at the last iteration; the refinement *it* = *count* in both *Hello* and *Finish* branches specifies a payload value *dependent* on the recursion variable *curr* and the variable *count* transmitted in the first message.

We establish the correctness of Refined MPST. In particular, we show that projection is behaviour-preserving and that well-formed global types with refinements satisfy progress, i.e. they do not get stuck. Therefore, if the endpoints follow the behaviour prescribed by the local types, derived (via projection) from a well-formed global type with refinements, the system is deadlock-free.

Callback-styled, Refinement-typed *APIs for Endpoint Implementations*. One of the main challenges in applying session types in practice is dealing with session *linearity*: a session channel is used *once and only once*. Session typing relies on a linear treatment of communication channels, in order to track the I/O actions performed on the channel against the intended session type. Most existing implementations adopt one of two approaches: monadic interfaces in functional languages [Imai et al. 2020, 2019; Orchard and Yoshida 2017], or "hybrid" approaches that complement static typing with dynamic linearity checks [Hu and Yoshida 2016; Scalas et al. 2017].

This paper proposes a fresh look to session-based programming that does *not* require linearity checking for static safety. We promote a form of session programming where session I/O is *implicitly* implemented by *callback functions* — we say "implicitly" because the user does not write any I/O operations themself: an *input callback* is written to *take* a received message as a parameter, and an *output callback* is written to simply *return* the label and value of the message to be sent.

The callbacks are supported by a runtime, generated along with APIs in refinement types according to the local type. The runtime performs communication and invokes user-specified callback functions upon corresponding communication events. We provide a code generation tool to streamline the writing of callback functions for the projected local type.

The inversion of control allows us to dispense with linearity checking, because our approach does not expose communication channels to the user. Our approach is a natural fit to functional programming settings, but also directly applicable to any statically typed language. Moreover, the linearity guarantee is achieved *statically* without the use of a linear type system, a feature that is usually not supported by mainstream programming languages. We follow the principle of event-based programming via the use of callbacks, prevalent in modern days of computing.

A Toolchain Implementation: Session*. To evaluate our proposal, we implement RMPST with a toolchain — Session*, as an extension to the Scribble toolchain [Hu 2017; Scribble Authors 2015] (http://www.scribble.org/) with F* [Swamy et al. 2016] as the target endpoint language.

Building on our callback approach, we show how to integrate RMPST with the verification-oriented functional programming language F*, exploiting its capabilities of refinement types and static verification to extend our fully static safety guarantees to data refinements in sessions. Our experience of specifying and implementing protocols drawn from the literature and real-world applications attests to the practicality of our approach and the value of statically verified refinements. Our integration of RMPST and F* allows developers to utilise advanced type system features to implement safe distributed application protocols.

Paper Summary and Contributions.

§ 2 We present an overview of our toolchain: Session*, and provide background knowledge of Scribble and MPST. We use a number guessing game, HigherLower, as our running example.

§ 3 We introduce Session*, a toolchain for RMPST. We describe in detail how our generated APIs can be used to implement multiparty protocols with refinements.

§ 4 We establish the metatheory of RMPST, which gives semantics of global and local types with refinements. We prove trace equivalence of global and local types w.r.t. projection (Theorem 4.10), and show progress and type safety of well-formed global types (Theorem 4.14 and Theorem 4.15).

§ 5 We evaluate our toolchain with examples from the session type literature, and measure the time taken for compilation and execution. We show that our toolchain does not have a long compilation time, and our runtime does not incur a large overhead on execution time.

The accompanying artifact [Zhou et al. 2020] contains the source code of our toolchain Session*, with examples and benchmarks used in the evaluation. The artifact is available as a Docker image, and can be accessed on the Docker Hub. The source files are available on GitHub (https://github.
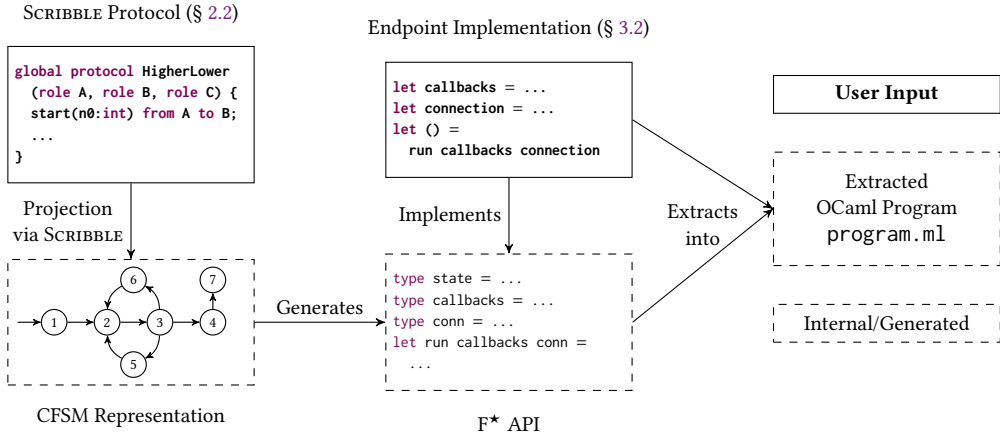
Fig. 2. Overview of Toolchain

com/sessionstar/oopsla20-artifact). We present the proof of our theorems, and additional technical details of the toolchain, in the full version of the paper (https://arxiv.org/abs/2009.06541).

2 OVERVIEW OF REFINED MULTIPARTY SESSION TYPES

In this section, we give an overview of our toolchain: SESSION*, describing its key toolchain stages. SESSION* extends the SCRIBBLE toolchain with refinement types and uses F* as a target language. We begin with a short background on basic multiparty session types and SCRIBBLE, then demonstrate the specification of distributed applications with refinements using the extended SCRIBBLE.

2.1 Toolchain Overview

We present an overview of our toolchain in Fig. 2, where we distinguish user provided input by developers in solid boxes, from generated code or toolchain internals in dashed boxes.

Development begins with *specifying a protocol* using an extended SCRIBBLE protocol description language. SCRIBBLE is closely associated with the MPST theory [Hu 2017; Neykova and Yoshida 2019], and provides a user-friendly syntax for multiparty protocols. We extend the SCRIBBLE toolchain to support RMPST, allowing refinements to be added via annotations. The extended SCRIBBLE toolchain (as part of SESSION*) validates the well-formedness of the protocol, and produces a representation in the form of a *communicating finite state machine* (CFSM, [Brand and Zafiropulo 1983]) for a given participant.

We then use a code generator (also as part of SESSION*) to generate F* APIs from the CFSM, utilising a number of advanced type system features available in F* (explained later in § 3.1). The generated APIs, described in detail in § 3, consist of various type definitions, and an entry point function taking *callbacks* and *connections* as arguments.

In our design methodology, we separate the concern of *communication* and *program logic*. The callbacks, corresponding to program logic, do not involve communication primitives — they are invoked to prompt a value to be sent, or to process a received value. Separately, developers provide a connection that allows base types to be serialised and transmitted to other participants. Developers implement the endpoint by providing both *callbacks* and *connections*, according to the generated refinement typed APIs. They can run the protocol by invoking the generated entry point. Finally, the F* source files can be verified using the F* compiler, and extracted to an OCaml program (or other supported targets) for efficient execution.

```
1  global protocol HigherLower(role A, role B, role C) {
2     // A tells B a secret number `n0`,
3     // and the number `t0` of attempts that C has to guess it
4     start(n0:int) from A to B;   @'0≤n0<100'
5     limit(t0:int) from A to B;   @'0<t0'
6     do Aux(A, B, C);             @'B[n0, t0]'  }
7  aux global protocol Aux(role A, role B, role C) @'B[n:int{0≤n<100}, t:int{0<t}]' {
8     guess(x:int)           from C to B;     @'0≤x<100'         // Next guess by C
9     choice at B {   higher() from B to C;   @'n>x ∧ t>1'       // Secret is higher
10                    higher() from B to A;
11                    do Aux(A, B, C);        @'B[n, t-1]'
12            } or {  win()    from B to C;    @'n=x'             // C wins, A loses
13                    lose()   from B to A;
14            } or {  lower()  from B to C;    @'n<x ∧ t>1'       // Secret is lower
15                    lower()  from B to A;
16                    do Aux(A, B, C);        @'B[n, t-1]'
17            } or {  lose()   from B to C;    @'n≠x ∧ t=1'       // A wins, C loses
18                    win()    from B to A;
19  }         }
```

Fig. 3. A *Refined* Scribble Global Protocol for a `HigherLower` Game.

2.2 Global Protocol Specification — RMPST in Extended Scribble

The workflow in the standard MPST theory [Honda et al. 2008], as is generally the case in distributed application development, starts from identifying the intended protocol for participant interactions. In our toolchain, a *global protocol*—the description of the whole protocol between all participants from a bird eye's view—is specified using our RMPST extension of the Scribble protocol description language [Hu 2017; Scribble Authors 2015]. Figure 3 gives the global protocol for a three-party game, `HigherLower`, which we use as a running example.

Basic Scribble/MPST. We first explain basic Scribble (corresponding to the standard MPST) *without* the @-annotations (annotations are extensions to the basic Scribble).

(1) The main protocol `HigherLower` declares three *roles* **A**, **B** and **C**, representing the runtime communication session participants. The protocol starts with **A** sending **B** a `start` message and a `limit` message, each carrying an `int` payload.

(2) The `do` construct specifies all three roles to proceed according to the (recursive) `Aux` sub-protocol. **C** sends **B** a `guess` message, also carrying an `int`. (The `aux` keyword simply tells Scribble that a sub-protocol does not need to be verified as a top-level entry protocol.)

(3) The `choice at B` construct specifies at this point that **B** should decide (make an *internal* choice) by which one of the four cases the protocol should proceed. This decision is explicitly communicated (as an *external* choice) to **A** and **C** via the messages in each case. The `higher` and `lower` cases are the recursive cases, leading to another round of `Aux` (i.e. another guess by **C**); the other two cases, `win` and `lose`, end the protocol.

To sum up, **A** sends **B** two numbers, and **C** sends a number (at least one) to **B** for as long as **B** replies with either `higher` or `lower` to **C** (and **A**). Next we demonstrate how we can express data dependencies using refinements with our extended Scribble.

Extended Scribble/RMPST. As described above, a basic global protocol (equivalent to a standard MPST *global type*) specifies the structure of participant interactions, but is not very informative

about the behaviour of the *application* in the aspect of data transmitted. This limitation can be contrasted against standardised (but informal) specifications of real-world application protocols, e.g. HTTP, OAuth, where a significant part, if not the majority, of the specification is devoted to the *data* side of the protocol. It goes without saying, details missing from the specification cannot be verified on an implementation.

We go through Fig. 3 again, this time including the practical extensions proposed by this paper to address these limitations: RMPST enables a refinement type–based treatment of protocols, capturing and integrating both data constraints and interactions. While refinement types by themselves can already greatly enrich the specification of individual messages, the most valuable and interesting aspect is the *interplay* between data refinements and the consequent interactions (i.e. the protocol flow) in the distributed, multiparty setting. In our setup, we allow data-dependent refinements to be specified in the global protocol, we explain various ways of using them in the running example:

- **Message Values.** A basic use of refinements is on message *values*, specifically their payload contents. The annotation on the first interaction (Line 4) specifies that **A** and **B** not only communicate an n0:int, but that 0≤n0<100 is a *post*condition for the value produced by **A** and a *pre*condition on the value received by **B**. Similarly, the int carried by limit must be positive.
- **Local Protocol State.** RMPST also supports refinements on the *recursions* that a protocol transitions through. The B[...] annotation (Line 7) in the Aux header specifies the *local* state known by B during the recursion, whenever **B** reaches *this point* in the ongoing protocol. The local state includes an int n such that 0≤n<100 and an int t such that 0<t. These extra variables are available for all enactments of this subprotocol. That is, on the first entry from HigherLower, where the do annotation B[n0, t0] (Line 6) specifies the initial values; and from the recursive entries (Line 11).

  By *known* state, we mean that **B** will have access to the exact values at runtime, although statically we can only be sure that they lie within the intervals specified in the refinements. Other session participants can only use the type information, without knowing the value, e.g. **C** does not know the exact value of n (which is the main point of this game), but knows the range via the refinements, and hence the endpoint may utilise this knowledge for reasoning.
- **Protocol Flow.** As mentioned, RMPST combines protocol specifications with refinements in order to direct the flow of the protocol — specifically at internal choices. The annotation on the win interaction (Line 12) from B to **C** specifies that B can only send this message, and thus select this choice case, after a correct guess by **C**. Similarly, **B** can only select the other cases after an incorrect guess: lose (Line 17) when **C** is on its last attempt, or the corresponding higher (Line 9) or lower (Line 14) cases otherwise. We exploit the fact that a refinement type can be uninhabited due to the impossibility to satisfy the constraint on the type, to encode protocol flow conditions.

Refinements allow a basic description of an interaction structure to be elaborated into a more expressive application specification. Note the t>1 in the higher and lower refinements are necessary to ensure that the 0<t in the Aux refinement is satisfied, given that the do annotations specify Aux to be recursively enacted with t-1. Albeit simple, the protocol shows how we can use refinements in various means to express data and control flow dependencies and recursive invariants in a multiparty setup. Once the protocol is specified, our toolchain allows the refinements in RMPST to be directly mapped to data refinements in F* through a callback-styled API generation.

3 IMPLEMENTING REFINED PROTOCOLS IN F⋆

In this section, we demonstrate our callback-styled, refinement-typed APIs for implementing endpoints in F⋆ [Swamy et al. 2016]. We introduced earlier the workflow of our toolchain (§ 2.1). In § 3.1, we summarise the key features of F⋆ utilised in our implementation. Using our running

example, we explain the generated APIs in F* in § 3.2, and how to implement endpoints in § 3.3. We outline the function we generate for executing the endpoint in § 3.4.

Developers using Session* implement the callbacks in F*, to utilise the functionality of refinement types provided by the F* type-checker. The F* compiler can verify statically whether the provided implementation by the developer satisfies the refinements as specified in the protocol. The verified code can be extracted via the F* compiler to OCaml and executed.

The verified implementation enjoys properties both from the MPST theory, such as *session fidelity* and *deadlock freedom*, but also from refinement types, that the data dependencies are verified to be correct according to the protocol. Additional details on code generation can be found in the full version of the paper.

3.1 Targeting F* and Implementing Endpoints

F* [Swamy et al. 2016] is a verification-oriented programming language, with a rich set of features. Our method of API generation and example programs utilise the following F* features:[1]

- **Refinement Types.** A *refinement type* has the form x:t{E}, where t is the base type, x is a variable that stands for values of this type, and E is a pure[2]boolean expression for *refinement*, possibly containing x. In short, the values of this refinement type are the *subset* of values of t that make E evaluate to true, e.g. natural numbers are defined as x:int{x≥0}. We use this feature to express data and control flow constraints in protocols.

  In F*, type-checking refinement types are done with the assistance of the Z3 SMT solver [De Moura and Bjørner 2008]. Refinements are encoded into SMT formulae and the solver decides the satisfiability of SMT formulae during type-checking. This feature enables automation in reasoning and saves the need for manual proofs in many scenarios.

- **Indexed Types.** *Types* can take pure expressions as arguments. For example, a declaration type t (i:t') = ... prescribes the *family* of types given by applying the type constructor t to *values* of type t'. We use this feature to generate type definitions for payload items in an internal choice, where the refinements in payload types refer to a state type.

- **Dependent Functions with Effects.** A (dependent) function in F* has a type of the form (x:t$_1$) → E t$_2$, where t$_1$ is the argument type, E describes the *effect* of the function, and t$_2$ is the result type, which may also refer to the argument x.

  The default effect is Tot, for pure total expressions (i.e. terminating and side-effect free). At the other end of the spectrum is the arbitrary effect ML (correspondent to all possible side effects in an ML language), which permits state mutation, non-terminating recursion, I/O, exceptions, etc.

- **The Ghost Effect and the erased Type.** A type can be marked erased in F*, so that values of such types are not available for computation (after extracting into target language), but only for proof purposes (during type-checking). The type constructor is accompanied with the Ghost effect to mark computationally irrelevant code, where the type system prevents the use of erased values in computationally relevant code, so that the values can really be safely erased. In the following snippet, we quickly demonstrate this feature: GTot stands for Ghost and total, and cannot be mixed with the default pure effect (the function not_allowed does not type-check). We use the erased type to mark variables known to the endpoint via the protocol specification, whose values are not known due to not being a party of the message interaction. For example, in Fig. 3, the endpoint C does not know the value of n0, but knowns its type from the protocol.

---

[1]A comprehensive F* tutorial is available at https://www.fstar-lang.org/tutorial/.
[2]Pure in this context means pure terminating computation, i.e. no side-effects including global state modifcations, I/O actions or infinite recursions, etc.

```
1  type t = { x1: int;                          1  (* The following access is not allowed *)
2             x2: erased int; }                 2  let not_allowed (o: t) = reveal o.x2
3  (* Definition in standard library *)         3  (* Accessing at type level is allowed *)
4  val reveal: erased a → GTot a                4  val allowed: (o: t{reveal o.x2 ≥ 0}) → int
```

Our generated code consists of multiple type definitions and an entry point function (as shown in Fig. 2, F⋆ API), including:

State Types: Allowing developers to access variables known at a given CFSM state.

Callbacks: A record of functions corresponding to CFSM transitions, used to implement program logic of the local endpoint.

Connections: A record of functions for sending and receiving values to and from other roles in the global protocol, used to handle the communication aspects of the local endpoint.

Entry Point: A function taking callbacks and connections to run the local endpoint.

To implement an endpoint, the developer needs to provide implementations for the generated callback and connection types, using appropriate functions to handle the program logic and communications. The F⋆ compiler checks whether the implemented functions type-check against the prescribed types. If the endpoint implementation succeeds the type-checking, the developer may choose to extract to a target language (e.g. OCaml, C) for execution.

3.2 Projection and F⋆ API Generation – Communicating Finite State Machine–based Callbacks for Session I/O

As in the standard MPST workflow, the next step (Fig. 2) is to *project* our *refined* global protocol onto each role. This decomposes the global protocol into a set of *local* protocols, prescribing the view of the protocol from each role. Projection is the definitive mechanism in MPST: although all endpoints together must comply to global protocol, projection allows each endpoint to be *separately* implemented and verified, a key requirement for practical distributed programming. As we shall see, the way projection treats refinements—we must consider the local *knowledge* of values propagated through the multiparty protocol—is crucial to verifying refined implementations, including our simple running example.

Projection onto B. We first look at the projection onto **B**: although it is the largest of the three projections, it is unique among them because **B** is involved in *every* interaction of the protocol, and (consequently) **B** has *explicit* knowledge of the value of *every* refinement variable during execution.

Formally, projection is defined as a syntactic function (explained in detail later in § 4.3); it is a partial function, designed conservatively to reject protocols that are not safely realisable in asynchronous distributed settings. However, we show in Fig. 4a the representation of projections employed in our toolchain based on *communicating finite state machines* (CFSMs) [Brand and Zafiropulo 1983], where the transitions are the localised I/O actions performed by **B** in this protocol. Projected CFSM actions preserve their refinements: as before, an action refinement serves as a precondition for an output transition to be fired, and a postcondition when an input transition is fired. For example, $A?\text{start}(n_0)\{0 \leq n_0 < 100\}$ is an input of a `start` message from **A**, with a refinement on the int payload value. Similarly, $C!\text{higher}\{n > x \land t > 1\}$ expresses a protocol flow refinement on an output of a `higher` message to **C**. For brevity, we omit the payload data types in the CFSM edges, as this example features only ints; we omit empty payloads "()" likewise.

We show the local state refinements as annotations on the corresponding CFSM states (shaded in grey, with an arrow to the state).

Refined API Generation for B. CFSMs offer an intuitive understanding of the semantics of endpoint projections. Building on recent work [Castro et al. 2019; Hu and Yoshida 2016; Neykova
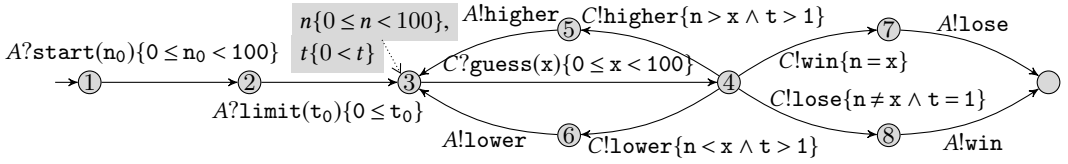
(a) CFSM Representation of the Projection. ! stands for sending actions, and ? for receiving actions on edges.

Generated F⋆ API

State	Edge	Generated Callback Type
1	A?start	s1 $\rightarrow$ (n:int{$0 \leq n < 100$}) $\rightarrow$ ML unit
2	A?limit	s2 $\rightarrow$ (t:int{$0 < t$}) $\rightarrow$ ML unit
3	C?guess	s3 $\rightarrow$ (x:int{$0 \leq x$}) $\rightarrow$ ML unit
4	[multiple]	(s:s4) $\rightarrow$ ML (s4Cases s)
5	A!higher	s5 $\rightarrow$ ML unit
6	A!lower	s6 $\rightarrow$ ML unit
7	A!lose	s7 $\rightarrow$ ML unit
8	A!win	s8 $\rightarrow$ ML unit

(b) Generated I/O Callback Types

```
type s4Cases (s:s4) =
| s4_lower of
  unit{s.n<s.x ∧ s.t>1}
| s4_lose of
  unit{s.n≠s.x ∧ s.t=1}
| s4_win of unit{s.n=s.x}
| s4_higher of
  unit{s.n>s.x ∧ s.t>1}
```

(c) Generated Data Type for the Output Choice

Fig. 4. Projection and F⋆ API Generation for **B** in HigherLower

et al. 2018], we use our CFSM-based representation of refined projections to *generate* protocol- and role-specific APIs for implementing each role in F⋆. We highlight a novel and crucial development: we exploit the approach of *type* generation to produce functional-style *callback*-based APIs that *internalise* all of the actual communication channels and I/O actions. In short, the transitions of the CFSM are rendered as a set of *transition-specific* function types to be implemented by the user — each of these functions take and return only the *user-level data* related to I/O actions and the running of the protocol. The transition function of the CFSM itself is embedded into the API by the generation, exporting a user interface to execute the protocol by calling back the appropriate user-supplied functions according to the current CFSM state and I/O event occurrences.

We continue with our example, Fig. 4b lists the function types for **B**, detailed as follows. Note, a characteristic of MPST-based CFSMs is that each non-terminal state is either input- or output-only.

- **State Types.** For each state, we generate a separate type (named by enumerating the states, by default). Each is defined as a record containing previously known payload values and its local recursion variables, or unit if none, for example:

    type s3 = { n0: int{$0 \leq n0 < 100$}; t0: int{$0 < t0$}; n: int{$0 \leq n < 100$}; t: int{$0 < t$} }

- **Basic I/O Callbacks.** For each input transition we generate a function type s $\rightarrow \sigma \rightarrow$ ML unit, where s is the predecessor state type, and σ is the refined payload type received. The return type is unit and the function can perform side effects, i.e. the callback is able to modify global state, interact with the console, etc, instead of merely pure computation. If an input transition is fired during execution, the generated runtime will invoke a user-supplied function of this type with the appropriately populated value of s, including any payload values received in the message that triggered this transition. Note, any data or protocol refinements are embedded into the types of these fields.

    Similarly, for each transition of an output state with a *single* outgoing transition, we generate a function type s $\rightarrow$ ML τ, where τ is the refined type for the output payload.

- **Internal Choices.** For each output state with more than 1 outgoing transition, we generate an additional sum type ρ with the cases of the choice, e.g. Fig. 4c. This sum type (s4Cases) is indexed

by the corresponding state type (s) to make any required knowledge available for expressing the protocol flow refinement of each case. Its constructors indicate the label of the internal choice. We then generate a single function type for this state, s → ML ρ: the user implementation selects which choice case to follow by returning a corresponding ρ value, under the constraints of any refinements. For example, a s4_win value can only be constructed, thus this choice case only be selected, when s.n=s.x for the given s. The state machine is advanced according to the constructor of the returned value (corresponding to the label of the message), and the generated runtime sends the payload value to the intended recipient.

An asynchronous *output* event, i.e. the trigger for the API to call back an output function, requires the communication medium to be ready to accept the message (e.g. there is enough memory in the local output buffer). For simplicity, in this work we consider the callbacks of an output state to always be immediately fireable. Concretely, we delegate these concerns to the underlying libraries and runtime system.

Projection and API Generation for C. The projection onto **C** raises an interesting question related to the refinement of *multiparty* protocols: how should we treat refinements on variables that the target role does *not* itself *know*? **C** does not know the value of the secret n (otherwise this game would be quite boring), but it does know that this information *exists* in the protocol and is subject to the specified refinement. In standard MPST, it is essentially the main point of projection that interactions not involving the target role can be simply (and safely) *dropped* outright; e.g. the communication of the start message would simply not appear in the projection of **C**. However, naively taking the same approach in RMPST would be inadequate: although the target role may not know some exact value, the role may still need the associated "latent information" to fulfil the desired application behaviour.

Our framework introduces a notion of *erased* variables for RMPST — in short, our projection does drop third-party *interactions*, but retains the latent information as refinement-typed *erased* variables, as illustrated by the annotation on state 1 in Fig. 5a. Thanks to the SMT-based refinement type system of F*, the type-checker can still take advatange of the refined types of erased variables to *prove* properties of the endpoint implementation; however, these variables cannot actually be used *computationally* in the implementation (since their values are not known). Conveniently, F* supports erased types (described briefly in § 3.1), and provides ways (i.e. Ghost effects) to ensure that such variables are not used in the computation. We demonstrate this for our example in the next subsection. Our approach can be considered a version of *irrelevant* variables from [Abel and Scherer 2012; Pfenning 2001] for the setting of typed, distributed interactions.

3.3 F* Implementation – Protocol Validation and Verification by Refinement Types

Finally, the generated APIs—embodying the refined projections—are used to implement the endpoint processes. As mentioned, the user implements the program logic as callback functions of the generated (refinement) types, supplied to the entry point along with code for establishing the communication channels between the session peers. Assuming a record callbacks containing the required functions (static typing ensures all are covered), Fig. 6a bootstraps a **C** endpoint.

The API takes care of endpoint execution by monitoring the channels, and calling the appropriate callback based on the current protocol state and I/O event occurrences. For example, a minimal, well-typed implementation of **B** could comprise the internal choice callback above (Fig. 6b) (implementing the type in Fig. 4c), cf. state 4, and an empty function for all others (i.e. fun _ → ()). We can highlight how protocol violations are ruled out by static refinement typing, which is ultimately the practical purpose of RMPST. In the above callback code, changing, say, the condition for the lose case to s.t=0 would directly violate the refinement on the s4_lose constructor, cf. Fig. 4c. Omitting
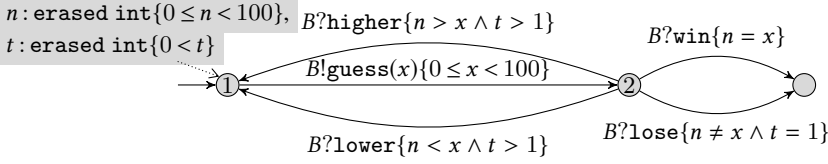
(a) CFSM Representation of the Projection

State	Edge	Generated type	User implementation
			(* Allocate a refined int reference *)
			`let next: ref (x:int{0≤x<100}) = alloc 50`
1	B!guess	`s1 → ML (x:int{0≤x<100})`	`fun _ → !next (*Deref next*)`
2	B?higher	`s2 → unit{n>x ∧ t>1} → ML unit`	`fun s → next := s.x + 1`
	B?lower	`s2 → unit{n<x ∧ t>1} → ML unit`	`fun s → next := s.x - 1`
	B?win	`s2 → unit{n=x} → ML unit`	`fun _ → ()`
	B?lose	`s2 → unit{n≠x ∧ t=1} → ML unit`	`fun _ → ()`

(b) Generated I/O Callback Types

Fig. 5. Projection and F⋆ API Generation for **C** in HigherLower

```
let main () =
  (* connect to B via TCP *)
  let server_B = connect ip_B port_B in
  (* Setup connection from TCP *)
  let conn = mk_conn server_B in
  (* Invoke the Entry Point `run` *)
  let () = run callbacks conn in
  (* Close TCP connection *)
  close server_B
```

```
(* Signature (s:s4) →  ML (s4Cases s) *)
fun (s:s4) →
  (* Win if guessed number is correct *)
  if s.x=s.n then s4_win ()
  (* Lose if running out of attempts *)
  else if s.t=1 then s4_lose ()
  (* Otherwise give hints accordingly *)
  else if s.n>s.x then s4_higher ()
  else s4_lower ()
```

(a) Running the Endpoint **C**   (b) Implementing the Internal Choice for **B**

Fig. 6. Selected Snippets of Endpoint Implementation

the lose case altogether would break both the lower and higher cases, as the type checker would not be able to prove s.t>1 as required by the subsequent constructors.

Lastly, Fig. 5b implements **C** to guess the secret by a simple search, given we know its value is bounded within the specified interval. We draw attention to the input callback for higher, where we adjust the next value. Given that the value being assigned is one more than the existing value, it might have been the case that the new value falls out of the range (in the case where next is 99), hence violating the prescribed type. However, despite that the value of n is unknown, we have known from the refinement attached to the edge that n>x holds, hence it must have been the case that our last guess x is strictly less than the secret n, which rules out the possibility that x can be 99 (the maximal value of n). Had the refinement and the erased variable not been present, the type-checker would not be able to accept this implementation, and it demonstrates that our encoding allows such reasoning with latent information from the protocol.

Moreover, the type and effect system of F⋆ prevents the erased variables from being used in the callbacks. On one hand, int and erased int types are not compatible, because they are not the same type. This prevents an irrelevant variable from being used in place of a concrete variable. On the other hand, the function reveal converts a value of erased 'a to a value of 'a with Ghost effect. A function with Ghost effect *cannot* be mixed with a function with ML effect (as in the case of our callbacks), so irrelevant variables cannot be used in the implementation via the reveal function.

Interested readers are invited to try the running example out with our accompanying artifact. We propose a few modifications on the implementation code and the protocol, and invite the readers to observe errors when implementations no longer correctly conforms to the prescribed protocol.

3.4 Executing the Communicating Finite State Machine (Generated Code)

As mentioned earlier, our API design sepearates the concern of program logic (with callbacks) and communication (with connections). A crucial piece of the generated code involves *threading* the two parts together — the execution function performs the communications actions and invokes the appropriate callbacks for handling. In this way, we do *not* expose explicit communication channels, so linearity can be achieved with ease by construction in our generated code.

The entry point function, named run, takes callbacks and connections as arguments, and executes the CFSM for the specified endpoint. The signature uses the permissive ML effect, since communicating with the external world performs side effects. We traverse the states (the set of states is denoted $\mathbb{Q}$) in the CFSM and generate appropriate code depending on the nature of the state and its outgoing transitions.

Internally, we define mutually recursive functions for each state $q \in \mathbb{Q}$, named run_q, taking the state record $[\![q]\!]$ as argument ($[\![q]\!]$ stands for the state record for a given state q), which performs the required actions at state q. The run state function for a state q either (1) invokes callbacks and communication primitives, then calls the run state function for the successor state q', or (2) returns directly for termination if q is a terminal state (without outgoing transitions). The main entry point invokes the run function for the initial state q_0, starting the finite state machine.

The internal run state functions are not exposed to the developer, hence it is not possible to tamper with the internal state with usual means of programming. This allows us to guarantee linearity of communication channels by construction. In the following text, we outline how to run each state, depending on whether the state is a sending state or a receiving state. Note that CFSMs constructed from local types do not have mixed states [Deniélou and Yoshida 2013, Prop. 3.1]

```
let rec run_q (st: stateq) =
  let choice = callbacks.stateq_send st
  in match choice with
  | Choiceqli payload →
    comm.send_string q "li";
    comm.send_S q payload;
    let st = { ···; xi=payload } in
    run_q' st
```
⎫
⎬ Repeat
⎭ for i ∈ I

```
let rec run_q (st: stateq) =
  let label = comm.recv_string p () in
  match label with
  | "li" →
    let payload = comm.recv_S p () in
    callbacks.stateq_receive_li st payload;
    let st = { ···; xi=payload } in
    run_q' st
```

(a) Template for Sending State q (b) Template for Receiving State q

Fig. 7. Template for run_q

Running the CFSM at a Sending State. For a sending state $q \in \mathbb{Q}$, the developer makes an internal choice on how the protocol proceeds, among the possible outgoing transitions. This is done by invoking the sending callback state$q$_send with the current state record, to obtain a choice with the associated payload. We pattern match on the constructor of the label l_i of the message, and find the corresponding successor state q'.

The label l_i is encoded as a string and sent via the sending primitive to q. It is followed by the payload specified in the return value of the callback, via corresponding sending primitive according to the base type with refinement erased.

We construct a state record of $[\![q']\!]$ from the existing record $[\![q]\!]$, adding the new field x_i in the action using the callback return value. In the case of recursive protocols, we also update the recursion variable according to the definition in the protocol when constructing $[\![q']\!]$. Finally, we call the run state function $\mathsf{run}_{q'}$ to continue the CFSM, effectively making the transition to state q'.

Following the procedure, run_q is generated as shown in Fig. 7a.

Running the CFSM at a Receiving State. For a receiving state $q \in \mathbb{Q}$, how the protocol proceeds is determined by an external choice, among the possible outgoint actions. To know what choice is made by the other party, we first receive a string and decode it into a label l, via the receiving primitive for string.

Subsequently, according to the label l, we can look up the label in the possible transitions, and find the state successor q'. By invoking the appropriate receiving primitive, we obtain the payload value. We note that the receiving primitive has a return type without refinements. In order to re-attach the refinements, we use the F* builtin `assume` to reinstate the refinements according to the protocol before using the value.

According the label l received, we can call the corresponding receiving callback with the received value. This allows the developer to process the received value and perform any relevant program logic. This is followed by the same procedure for constructing the state record for the next state q' and invoking the run function for q'.

Following the procedure, run_q is generated as shown in Fig. 7b.

3.5 Summary

We demonstrated with our running example, `HigherLower`, how to implement a refined multiparty protocol with our toolchain Session*.

Exploiting the powerful type system of F*, our approach has several key benefits: First, it guarantees *fully static* session type safety in a lightweight, practical manner — the callback-style API is portable to any statically typed language. Existing work based on code generation has considered only hybrid approaches that supplement static typing with dynamically checked linearity of explicit communication channel usages. Moreover, the separation of program logic and communication leads to a modular implementation of protocols.

Second, it is well suited to functional languages like F*; in particular, the *data*-oriented nature of the user interface allows the refinements in RMPST to be directly mapped to data refinements in F*, allowing the refinements constraints to be discharged at the user implementation level by the F* compiler — again, fully statically.

Furthermore, our endpoint implementation inherits core communication safety properties such as freedom from deadlock or communication mismatches, based on the original MPST theory. We use the F* type-checker to validate statically that an endpoint implementation is correctly typed with respect to the prescribed type obtained via projection of the global protocol. Therefore, the implementation benefits from additional guarantees from the refinement types.

4 A THEORY OF REFINED MULTIPARTY SESSION TYPES (RMPST)

In this section, we introduce *refined multiparty session types* (*RMPST* for short). We give the syntax of types in § 4.1, extending original multiparty session types (MPST) with *refinement types*. We describe the refinement typing system that we use to type expressions in RMPST in § 4.2.

We follow the standard MPST methodology. *Global session types* describe communication structures of many *participants* (also known as *roles*). *Local session types*, describing communication structures of a single participant, can be obtained via *projection* (explain in § 4.3). Endpoint processes implement local types obtained from projection. We give semantics of global types and local types

in § 4.4, and show the equivalence of semantics with respect to projection. As a consequence, we can compose all endpoint processes implementing local types for roles in a global type, obtained via projection, to implement the global type correctly.

4.1 Syntax of Types

We define the syntax of refined multiparty session types (refined MPST) in Fig. 8. We use different colours for different syntactical categories to help disambiguation, but the syntax can be understood without colours. We use pink for global types, dark blue for local types, blue for expressions, purple for base types, indigo for labels, and **Teal** with bold fonts for participants.

$$
\begin{array}{llll}
S & ::= \text{int} \mid \text{bool} \mid \dots & \text{Base Types} \\
T & ::= x : S\{E\} & \text{Refinement Types} \\
E & ::= x \mid \underline{n} \mid op_1 \; E \mid E \; op_2 \; E \dots & \text{Expressions} \\
G & ::= & \text{Global Types} \\
& \mid \quad \mathbf{p} \to \mathbf{q} \; \{l_i(x_i : T_i).G_i\}_{i \in I} & \text{Message} \\
& \mid \quad \mu t(x : T)\langle x := E\rangle.G & \text{Recursion} \\
& \mid \quad t\langle x := E\rangle \mid \text{end} & \text{Type Var., End}
\end{array}
\qquad
\begin{array}{lll}
L & ::= & \text{Local Types} \\
& \mid \quad \mathbf{p}\&\{l_i(x_i : T_i).L_i\}_{i \in I} & \text{Receiving} \\
& \mid \quad \mathbf{p}\oplus\{l_i(x_i : T_i).L_i\}_{i \in I} & \text{Sending} \\
& \mid \quad l(x : T).L & \text{Silent Prefix} \\
& \mid \quad \mu t(x : T)\langle x := E\rangle.L & \text{Recursion} \\
& \mid \quad t\langle x := E\rangle \mid \text{end} & \text{Type Var., End}
\end{array}
$$

Fig. 8. Syntax of Refined Multiparty Session Types

Value Types and Expressions. We use S for base types of values, ranging over integers, booleans, etc. Values of the base types must be able to be communicated.

The base type S can be *refined* by a boolean expression, acting as a predicate on the members of the base type. A *refinement type* is of the form $(x : S\{E\})$. A value x of the type has base type S, and is refined by a boolean expression E. The boolean expression E acts as a predicate on the members x (possibly involving the variable x). For example, we can express natural numbers as $(x : \text{int}\{x \geq 0\})$. We use $\text{fv}(\cdot)$ to denote the free variables in refinement types, expressions, etc. We consider variable x be bound in the refinement expression E, i.e. $\text{fv}(x : S\{E\}) = \text{fv}(E) \setminus \{x\}$.

Where there is no ambiguity, we use the base type S directly as an abbreviation of a refinement type $(x : S\{\text{true}\})$, where x is a fresh variable, and true acts as a predicate that accepts all values.

Global Session Types. Global session types (*global types* or *protocols* for short) range over $G, G', G_i, \dots$ Global types give an overview of the overall communication structure. We extend the standard global types [Deniélou and Yoshida 2013] with refinement types and variable bindings in message prefixes. Extensions to the syntax are shaded in the following explanations.

$\mathbf{p} \to \mathbf{q} \; \big\{l_i(\,\boxed{x_i : T_i}).G_i\big\}_{i \in I}$ is a message from $\mathbf{p}$ to $\mathbf{q}$, which branches into one or more continuations with label l_i, carrying a payload variable x_i with type T_i. We omit the curly braces when there is only one branch, like $\mathbf{p} \to \mathbf{q} : l(x : T)$. We highlight the difference from the standard syntax, i.e. the variable binding. The payload variable x_i occurs bound in the continuation global type G_i, for all $i \in I$. We sometimes omit the variable if it is not used in the continuations. The free variables are defined as:

$$
\text{fv}(\mathbf{p} \to \mathbf{q} \; \{l_i(x_i : T_i).G_i\}_{i \in I}) = \bigcup_{i \in I} \text{fv}(T_i) \cup \bigcup_{i \in I} (\text{fv}(G_i) \setminus \{x_i\})
$$

We require that the index set I is not empty, and all labels l_i are distinct. To prevent duplication, we write $l(x : S\{E\})$ instead of $l(x : (x : S\{E\}))$ (the first x occurs as a variable binding in the message, the second x occurs as a variable representing member values in the refinement types).

We extend the construct of recursive protocols to include a variable carrying a value in the inner protocol. In this way, we enhance the expressiveness of the global types by allowing a recursion variable to be maintained across iterations of global protocols.

The recursive global type $\mu t(\boxed{x:T})\langle\boxed{x := E}\rangle.G$ specifies a variable x carrying type T in the recursive type, initialised with expression E. The type variable $t\langle\boxed{x := E}\rangle$ is annotated with an assignment of expression E to variable x. The assignment updates the variable x in the current recursive protocol to expression E. The free variables in recursive type is defined as

$$\text{fv}(\mu t(x:T)\langle x := E\rangle.G) = \text{fv}(T) \cup \text{fv}(E) \cup (\text{fv}(G) \setminus \{x\})$$

We require that recursive types are contractive [Pierce 2002, §21], so that recursive protocols have at least a message prefix, and protocols such as $\mu t(x:T)\langle x := E_1\rangle.t\langle x := E_2\rangle$ are not allowed. We also require recursive types to be closed with respect to type variables, e.g. protocols such as $t\langle x := E\rangle$ alone are not allowed.

We write $G[\mu t(x:T).G/t]$ to substitute all occurrences of type variables with expressions $t\langle x := E\rangle$ into $\mu t(x:T)\langle x := E\rangle.G$. We write $r \in G$ to say r is a participating role in the global type G.

Example 4.1 (Global Types). We give the following examples of global types.

(1) $G_1 = A \rightarrow B : Fst(x:\text{int}).B \rightarrow C : Snd(y:\text{int}\{x = y\}).C \rightarrow D : Trd(z:\text{int}\{x = z\}).\text{end}$.

G_1 describes a protocol where A sends an int to B, and B relays the same int to C, similar for C to D. Note that we can write $x = z$ in the refinement of z, whilst x is not known to C.

(2) $G_2 = A \rightarrow B : Number(x:\text{int}).B \rightarrow C \begin{cases} Positive(\text{unit}\{x > 0\}).\text{end} \\ Zero(\text{unit}\{x = 0\}).\text{end} \\ Negative(\text{unit}\{x < 0\}).\text{end} \end{cases}$

G_2 describes a protocol where A sends an int to B, and B tells C whether the int is positive, zero, or negative. We omit the variable here since it is not used later in the continuation.

(3) $G_3 = \mu t(try:\text{int}\{try \geq 0 \wedge try \leq 3\})\langle try := 0\rangle.$
$\quad A \rightarrow B : Password(pwd:\text{string}).$
$\quad B \rightarrow A \begin{cases} Correct(\text{unit}).\text{end} \\ Retry(\text{unit}\{try < 3\}).t\langle try := try + 1\rangle \\ Denied(\text{unit}\{try = 3\}).\text{end} \end{cases}$

G_3 describes a protocol where A authenticates with B with maximum 3 tries.

Local Session Types. Local session types (*local types* for short) range over $L, L', L_i, \ldots$. Local types give a view of the communication structure of an endpoint, usually obtained from a global type. In addition to standard syntax, the recursive types are similarly extended as those of global types.

Suppose the current role is q, the local type $p \oplus \{l_i(x_i : T_i).L_i\}_{i \in I}$ describes that the role q sends a message to the partner role p with label l_i (where i is selected from an index set I), carrying payload variable x_i with type T_i, and continues with L_i. It is also said that the role q takes an *internal choice*. Similarly, the local type $p\&\{l_i(x_i : T_i).L_i\}_{i \in I}$ describes that the role q receives a message from the partner role p. In this case, it is also said that the role q offers an *external choice*. We omit curly braces when there is only a single branch (as is done for global messages).

We add a new syntax construct of $l(x:T).L$ for *silent local types*. We motivate this introduction of the new prefix to represent knowledge obtained from the global protocol, but not in the form of a message. Silent local types are useful to model variables obtained with irrelevant quantification [Abel and Scherer 2012; Pfenning 2001]. These variables can be used in the construction of a type, but cannot be used in that of an expression, as we explain later in § 4.2. We show an example of a silent local type later in Example 4.3, after we define *endpoint projection*, the process of obtaining local types from a global type.

$$[\text{WF-R}\textsc{ty}]$$
$$\frac{\Sigma^+, x : S \vdash E : \texttt{bool}}{\Sigma \vdash (x : S\{E\}) \; ty}$$

$$[\text{TE-V}\textsc{ar}]$$
$$\frac{}{\Sigma_1, x^\omega : T, \Sigma_2 \vdash x : T}$$

$$[\text{TE-P}\textsc{lus}]$$
$$\frac{\Sigma \vdash E_1 : \texttt{int} \qquad \Sigma \vdash E_2 : \texttt{int}}{\Sigma \vdash E_1 + E_2 : (v : \texttt{int}\{v = E_1 + E_2\})}$$

$$[\text{TE-S}\textsc{ub}]$$
$$\frac{\Sigma \vdash E : (v : S\{E_1\}) \qquad \text{Valid}(\llbracket \Sigma \rrbracket \wedge \llbracket E_1 \rrbracket \implies \llbracket E_2 \rrbracket)}{\Sigma \vdash E : (v : S\{E_2\})}$$

$$[\text{TE-C}\textsc{onst}]$$
$$\frac{}{\Sigma \vdash \underline{n} : (v : \texttt{int}\{v = \underline{n}\})}$$

Fig. 9. Selected Typing Rules for Expressions in a Local Typing Context

4.2 Expressions and Typing Expressions

We use E, E', E_i to range over expressions. Expressions consist of variables x, constants (e.g. integer literals $\underline{n}$), and unary and binary operations. We use an SMT assisted refinement type system for typing expressions, in the style of [Rondon et al. 2008]. The simple syntax of expressions allows all expressions to be encoded into SMT logic, for deciding a semantic subtyping relation of refinement types [Bierman et al. 2012].

Typing Contexts. We define two categories of typing contexts, for use in handling global types and local types respectively.

$$\Gamma ::= \varnothing \mid \Gamma, x^{\mathbb{P}} : T \qquad \Sigma ::= \varnothing \mid \Sigma, x^\theta : T \qquad \theta ::= 0 \mid \omega$$

We annotate global and local typing contexts differently. For global contexts Γ, variables carry the annotation of a set of roles $\mathbb{P}$, to denote the set of roles that have the knowledge of its value.

For local contexts Σ, variables carry the annotation of their multiplicity θ. A variable with multiplicity 0 is an irrelevantly quantified variable (irrelevant variable for short), which cannot appear in the expression when typing (also denoted as $x \div T$ in the literature [Abel and Scherer 2012; Pfenning 2001]). Such a variable can only appear in an expression used as a predicate, when defining a refinement type. A variable with multiplicity ω is a variable without restriction. We often omit the multiplicity ω.

Well-formedness. Since a refinement type can contain free variables, it is necessary to define well-formedness judgements on refinement types, and henceforth on typing contexts.

We define Σ^+ to be the local typing context where all irrelevant variables x^0 become unrestricted x^ω, i.e. $(\varnothing)^+ = \varnothing; (\Sigma, x^\theta : T)^+ = \Sigma^+, x^\omega : T$.

We show the well-formedness judgement of a refinement type [WF-R$\textsc{ty}$] in Fig. 9. For a refinement type $(x : S\{E\})$ to be a well-formed type, the expression E must have a boolean type under the context Σ^+, extended with variable x (representing the members of the type) with type S. The typing context Σ^+ promotes the irrelevant quantified variables into unrestricted variables, so they can be used in the expression E inside the refinement type.

The well-formedness of a typing context is defined inductively, requiring all refinement types in the context to be well-formed. We omit the judgements for brevity.

Typing Expressions. We type expressions in local contexts, forming judgements of form $\Sigma \vdash E : T$, and show key typing rules in Fig. 9. We modify the typing rules in a standard refinement type system [Rondon et al. 2008; Vazou et al. 2014, 2017], adding distinction between irrelevant and unrestricted variables.

[TE-C$\textsc{onst}$] gives constant values in the expression a refinement type that only contains the constant value. Similarly, [TE-P$\textsc{lus}$] gives typing derivations for the plus operator, with a corresponding refinement type encoding the addition.

$$\Gamma \cup \{x^{\mathbb{P}} : T\} = \begin{cases} \Gamma, x^{\mathbb{P}} : T & \text{if } x \notin \Gamma \\ \Gamma_1, x^{\mathbb{P}} : T, \Gamma_2 & \text{if } \Gamma = \Gamma_1, x^{\varnothing} : T, \Gamma_2 \\ \Gamma_1, x^{\mathbb{P}} : T, \Gamma_2 & \text{if } \Gamma = \Gamma_1, x^{\mathbb{P}} : T, \Gamma_2 \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \Sigma \cup \{x^{\theta} : T\} = \begin{cases} \Sigma, x^{\theta} : T & \text{if } x \notin \Sigma \\ \Sigma_1, x^{\theta} : T, \Sigma_2 & \text{if } \Sigma = \Sigma_1, x^0 : T, \Sigma_2 \\ \Sigma_1, x^{\omega} : T, \Sigma_2 & \text{if } \Sigma = \Sigma_1, x^{\omega} : T, \Sigma_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 10. Typing Context Extension

We draw attention to the handling of variables ([TE-Var]). An irrelevant variable in the typing context cannot appear in an expression, i.e. there is *no* derivation for $\Sigma_1, x^0 : T, \Sigma_2 \vdash x : T$. These variables can only be used in an refinement type (see [WF-Rty]).

The key feature of the refinement type system is the semantic subtyping relation decided by SMT [Bierman et al. 2012], we describe the feature in [TE-Sub]. We use $[\![E]\!]$ to denote the encoding of expresion E into the SMT logic. We encode a type binding $x^{\theta} : (v : S\{E\})$ in a typing context by encoding the term $E[x/v]$, and define the encoding of a typing context $[\![\Sigma]\!]$ inductively.

We define the extension of typing contexts ($\Gamma \cup \{x^{\mathbb{P}} : T\}$; $\Sigma \cup \{x^{\theta} : T\}$) in Fig. 10, used in definitions of semantics. We say a global type G (resp. a local type L) is closed under a global context Γ (resp. a local context Σ), if all free variables in the type are in the domain of the context.

Remark 4.2 (Empty Type). A refinement type may be *empty*, with no inhabited member.

We can construct such a type under the empty context $\varnothing$ as $(x : S\{\text{false}\})$ with any base types S. A more specific example is a refinement type for an integer that is both negative and positive ($x : \text{int}\{x > 0 \wedge x < 0\}$). Similarly, under the context $x^{\omega} : \text{int}\{x > 0\}$, the refinement type $y : \text{int}\{y < 0 \wedge y > x\}$ is empty. In these cases, the typing context with the specified type becomes inconsistent, i.e. the encoded context gives a proof of falsity.

Moreover, an empty type can also occur without inconsistency. For instance, in a typing context of $x^0 : \text{int}$, the type $y : \text{int}\{y > x\}$ is empty — it is not possible to produce such a value without referring to x (cf. [TE-Var]).

4.3 Endpoint Projection: From Global Contexts and Types to Local Contexts and Types

In the methodology of multiparty session types, developers specify a global type, and obtain local types for the participants via *endpoint projection* (*projection* for short). In the original theory, projection is a *partial* function that takes a global type G and a participant $\mathbf{p}$, and returns a local type L. The resulting local type L describes a the local communication behaviour for participant $\mathbf{p}$ in the global scenario. Such workflow has the advantage that each endpoint can obtain a local type separately, and implement a process of the given type, hence providing modularity and scalability.

Projection is defined as a *partial* function, since only *well-formed* global types can be projected to all participants. In particular, a *partial* merge operator $\sqcup$ is used during the projection, for creating a local type $\Sigma \vdash L_1 \sqcup L_2 = L_{\text{merged}}$ that captures the behaviour of two local types, under context Σ.

In RMPST, we first define the projection of global typing contexts (Fig. 11), and then define the projection of global types under a global typing context (Fig. 12). We use expression typing judgements in the definition of projection, to type-check expressions against their prescribed types.

Projection of Global Contexts. We define the judgement $\Gamma \upharpoonright \mathbf{p} = \Sigma$ for the projection of global typing context Γ to participant $\mathbf{p}$ in Fig. 11. In the global context Γ, a variable x is annotated with the set of participants $\mathbb{P}$ who know the value. If the projected participant $\mathbf{p}$ is in the set $\mathbb{P}$, [P-Var-ω] is applied to obtain an unrestricted variable in the resulting local context; Otherwise, [P-Var-0] is applied to obtain an irrelevant variable.

Projection of Global Types with a Global Context. When projecting a global type G, we include a global context Γ, forming a judgement of form $\langle \Gamma \prec G \rangle \upharpoonright \mathbf{p} = \langle \Sigma \prec L \rangle$. Projection

$$\boxed{\Gamma \upharpoonright \mathbf{p} = \Sigma}$$

$$[\text{P-Empty}] \quad \frac{}{\varnothing \upharpoonright \mathbf{p} = \varnothing}$$

$$[\text{P-Var-}\omega] \quad \frac{\mathbf{p} \in \mathbb{P} \qquad \Gamma \upharpoonright \mathbf{p} = \Sigma}{\Gamma, x^{\mathbb{P}} : T \upharpoonright \mathbf{p} = \Sigma, x^{\omega} : T}$$

$$[\text{P-Var-0}] \quad \frac{\mathbf{p} \notin \mathbb{P} \qquad \Gamma \upharpoonright \mathbf{p} = \Sigma}{\Gamma, x^{\mathbb{P}} : T \upharpoonright \mathbf{p} = \Sigma, x^{0} : T}$$

Fig. 11. Projection Rules for Global Contexts

$$[\text{P-Send}] \quad \frac{\Gamma \upharpoonright \mathbf{p} = \Sigma \qquad \forall i \in I. \qquad \Sigma \vdash T_i \ ty \qquad \langle \Gamma \cup \{x_i^{\{\mathbf{p}, \mathbf{q}\}} : T_i\} \lessdot G_i \rangle \upharpoonright \mathbf{p} = \langle \Sigma_i \lessdot L_i \rangle}{\langle \Gamma \lessdot (\mathbf{p} \to \mathbf{q} \ \{l_i(x_i : T_i).G_i\}_{i \in I}) \rangle \upharpoonright \mathbf{p} = \langle \Sigma \lessdot \mathbf{q} \oplus \{l_i(x_i : T_i).L_i\}_{i \in I} \rangle}$$

$$[\text{P-Recv}] \quad \frac{\Gamma \upharpoonright \mathbf{q} = \Sigma \qquad \forall i \in I. \qquad \Sigma \vdash T_i \ ty \qquad \langle \Gamma \cup \{x_i^{\{\mathbf{p}, \mathbf{q}\}} : T_i\} \lessdot G_i \rangle \upharpoonright \mathbf{q} = \langle \Sigma_i \lessdot L_i \rangle}{\langle \Gamma \lessdot (\mathbf{p} \to \mathbf{q} \ \{l_i(x_i : T_i).G_i\}_{i \in I}) \rangle \upharpoonright \mathbf{q} = \langle \Sigma \lessdot \mathbf{p} \& \{l_i(x_i : T_i).L_i\}_{i \in I} \rangle}$$

$$[\text{P-Phi}] \quad \frac{\Gamma \upharpoonright \mathbf{r} = \Sigma \quad \mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\} \quad \forall i \in I. \quad \Sigma \vdash T_i \ ty \quad \langle \Gamma \cup \{x_i^{\{\mathbf{p}, \mathbf{q}\}} : T_i\} \lessdot G_i \rangle \upharpoonright \mathbf{r} = \langle \Sigma_i \lessdot L_i \rangle}{\langle \Gamma \lessdot (\mathbf{p} \to \mathbf{q} \ \{l_i(x_i : T_i).G_i\}_{i \in I}) \rangle \upharpoonright \mathbf{r} = \langle \Sigma \lessdot \sqcup_{i \in I} l_i(x_i : T_i).L_i \rangle}$$

$$[\text{P-Rec-In}] \quad \frac{\Gamma \upharpoonright \mathbf{r} = \Sigma \quad \mathbf{r} \in G \quad \langle \Gamma \cup \{x^{\{\mathbf{r} | \mathbf{r} \in G\}} : T\} \lessdot G \rangle \upharpoonright \mathbf{r} = \langle \Sigma' \lessdot L \rangle \quad \Sigma \vdash T \ ty \quad \Sigma \vdash E : T}{\langle \Gamma \lessdot \mu t(x : T)\langle x := E \rangle.G) \rangle \upharpoonright \mathbf{r} = \langle \Sigma \lessdot \mu t \ (x : T)\langle x := E \rangle.L \rangle}$$

$$[\text{P-Rec-Out}] \quad \frac{\Gamma \upharpoonright \mathbf{r} = \Sigma \qquad \mathbf{r} \notin G}{\langle \Gamma \lessdot \mu t(x : T)\langle x := E \rangle.G) \rangle \upharpoonright \mathbf{r} = \langle \Sigma \lessdot \text{end} \rangle} \qquad [\text{P-End}] \quad \frac{\Gamma \upharpoonright \mathbf{r} = \Sigma}{\langle \Gamma \lessdot \text{end} \rangle \upharpoonright \mathbf{r} = \langle \Sigma \lessdot \text{end} \rangle}$$

$$\boxed{\langle \Gamma \lessdot G \rangle \upharpoonright \mathbf{p} = \langle \Sigma \lessdot L \rangle} \qquad [\text{P-Var}] \quad \frac{\Gamma \upharpoonright \mathbf{r} = \Sigma = \Sigma_1, x : T, \Sigma_2 \qquad \Sigma_1, x : T, \Sigma_2 \vdash E : T}{\langle \Gamma \lessdot t\langle x := E \rangle \rangle \upharpoonright \mathbf{r} = \langle \Sigma \lessdot t\langle x := E \rangle \rangle}$$

Fig. 12. Projection Rules for Global Types

rules are shown in Fig. 12. Including a typing context allows us to type-check expressions during projection, hence ensuring that variables attached to recursive protocols are well-typed.

If the prefix of G is a message from role $\mathbf{p}$ to role $\mathbf{q}$, the projection results a local type with a send (resp. receive) prefix into role $\mathbf{p}$ (resp. $\mathbf{q}$) via [P-Send] (resp. [P-Recv]). For other roles $\mathbf{r}$, the projection results in a local type with a *silent label* via [P-Phi], with prefix $l(x : T)$. This follows the concept of a coordinated distributed system, where all the processes follow a global protocol, and base assumptions of their local actions on actions of other roles not involving them. The projection defined in the original MPST theory does not contain information for role $\mathbf{r}$ about a message between $\mathbf{p}$ and $\mathbf{q}$. We use the silent prefix to retain such information, especially the refinement type T of the payload. For merging two local types (as used in [P-Phi]), we use a simple plain merge operator defined as $\Sigma \vdash L \sqcup L = L$, requiring two local types to be identical in order to be merged.[3]

If the prefix of G is a recursive protocol $\mu t(x : T)\langle x := E \rangle.G$, the projection preserves the recursion construct if the projected role is in the inner protocol via [P-Rec-In] and that the expression E can be typed with type T under the projected local context. Typing expressions under local contexts ensures that no irrelevant variables x^0 are used in the expression E, as no typing derivation exists for irrelevant variables. Otherwise projection results in end via [P-Rec-Out]. If G is a type variable $t\langle x := E \rangle$, we similarly validate that the expression E carries the specified type in the correspondent recursion definition, and its projection also preserves the type variable construct.

---

[3]We build upon the standard MPST theory with plain merging. Full merge [Denielou et al. 2012], allowing certain different index sets to be merged, is an alternative, more permissive merge operator. Our implementation Session* uses the more permissive merge operator for better expressiveness.

Example 4.3 (Projection of Global Types of Example 4.1 (1)). We draw attention to the projection of G_1 to **C**, under the empty context $\varnothing$.

$$\langle \varnothing \prec G_1 \rangle \upharpoonright \mathbf{C} = \langle \varnothing \prec Fst(x : \mathtt{int}).\mathbf{B}\&Snd(y : \mathtt{int}\{x = y\}).\mathbf{D}\oplus Trd(z : \mathtt{int}\{x = z\}).\mathsf{end} \rangle$$

We note that the local type for **C** has a silent prefix $Fst(x : \mathtt{int})$, which binds the variable x in the continuation. The silent prefix adds the variable x and its type to the "local knowledge" of the endpoint **C**, yet the actual value of x is unknown.

REMARK 4.4 (EMPTY SESSION TYPE). Global types G and local types L can be empty because one of the value types in the protocol in an empty type (cf. Remark 4.2).

For example, the local type $\mathbf{A}\oplus Impossible(x : \mathtt{int}\{x > 0 \land x < 0\}).\mathsf{end}$ cannot be implemented, since such an x cannot be provided.

For the same reason, the local type $Pos(x : \mathtt{int}\{x > 0\}).\mathbf{A}\oplus Impossible(y : \mathtt{int}\{y > x\}).\mathsf{end}$ cannot be implemented.

REMARK 4.5 (IMPLEMENTABLE SESSION TYPES). Consider the following session type:

$$L = \mathbf{B}\&Num(x : \mathtt{int}).\mathbf{B}\oplus \begin{cases} Pos(\mathtt{unit}\{x > 0\}).\mathsf{end} \\ Neg(\mathtt{unit}\{x < 0\}).\mathsf{end} \end{cases}.$$

When the variable x has the value 0, neither of the choices *Pos* or *Neg* could be selected, as the refinements are not satisfied. In this case, the local type L cannot be implemented, as the internal choice callback may not be implemented in a *total* way, i.e. the callback returns a choice label for all possible inputs of integer x.[4]

4.4 Labelled Transition System (LTS) Semantics

We define the labelled transition system (LTS) semantics for global types and local types. We show the trace equivalence of a global type and the collection of local types projected from the global type, to demonstrate that projection preserves LTS semantics. The equivalence result allows us to use the projected local types for the implementation of local roles separately. Therefore, we can implement the endpoints in $\mathrm{F}^\star$ separately, and they compose to the specified protocol.

We also prove a type safety result that well-formed global types cannot be stuck. This, combined with the trace equivalence result, guarantees that endpoints are free from deadlocks.

Actions. We begin with defining actions in the LTS system. We define the label in the LTS as $\alpha ::= \mathbf{p} \to \mathbf{q} : l(x : T)$, a message from role $\mathbf{p}$ to $\mathbf{q}$ with label l carrying a value named x with type T. We define $\mathrm{subj}(\alpha) = \{\mathbf{p}, \mathbf{q}\}$ to be the subjects of the action α, namely the two roles in the action.

Semantics of Global Types. We define the LTS semantics of global types in Fig. 13. Different from the original LTS semantics in [Deniélou and Yoshida 2013], we include the context Γ in the semantics along with the global type G. Therefore, the judgements of global LTS reduction have form $\langle \Gamma \prec G \rangle \xrightarrow{\alpha} \langle \Gamma' \prec G' \rangle$.

[G-PFX] allows the reduction of the prefix action in a global type. An action, matching the definition in set defined in the prefix, allows the correspondent continuation to be selected. The resulting global type is the matching continuation and the resulting context contains the variable binding in the action.

[G-CNT] allows the reduction of an action that is causally independent of the prefix action in a global type, here, the subjects of the action are disjoint from the prefix of the global type. If all continuations in the global types can make the reduction of that action to the same context, then

---

[4]Since we use a permissive ML effect in the callback type, allowing all side effects to be performed in the callback, the callback may throw exceptions or diverge in case of unable to return a value.

$$[\text{G-Pfx}] \quad \cfrac{j \in I}{\langle \Gamma \prec \mathbf{p} \to \mathbf{q}\, \{l_i(x_i : T_i).G_i\}_{i \in I}\rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:l_j(x_j:T_j)} \langle \Gamma \cup \{x_j^{\{\mathbf{p},\,\mathbf{q}\}} : T_j\} \prec G_j\rangle}$$

$$[\text{G-Cnt}] \quad \cfrac{\{\mathbf{p},\mathbf{q}\} \cap \mathrm{subj}(\alpha) = \varnothing \qquad \forall j \in I.\langle \Gamma \cup \{x_j^\varnothing : T_j\} \prec G_j\rangle \xrightarrow{\alpha} \langle \Gamma' \prec G_j'\rangle}{\langle \Gamma \prec \mathbf{p} \to \mathbf{q}\, \{l_i(x_i : T_i).G_i\}_{i \in I}\rangle \xrightarrow{\alpha} \langle \Gamma' \prec \mathbf{p} \to \mathbf{q}\, \{l_i(x_i : T_i).G_i'\}_{i \in I}\rangle}$$

$$[\text{G-Rec}] \quad \cfrac{\langle \Gamma \cup \{x^{\{\mathbf{r}\,|\,\mathbf{r}\in G\}} : T\} \prec G[\mu t(x:T).G/t]\rangle \xrightarrow{\alpha} \langle \Gamma' \prec G'\rangle}{\langle \Gamma \prec \mu t(x:T)\langle x := E\rangle.G\rangle \xrightarrow{\alpha} \langle \Gamma' \prec G'\rangle}$$

$$\boxed{\langle \Gamma \prec G\rangle \xrightarrow{\alpha} \langle \Gamma' \prec G'\rangle}$$

Fig. 13. LTS Semantics for Global Types

the result context is that context and the result global type is one with continuations after reduction. When reducing the continuations, we add the variable of the prefix action into the context, but tagged with an empty set of known roles. This addition ensures that relevant information obtainable from the prefix message is not lost when performing reduction.

[G-Rec] allows the reduction of a recursive type by unfolding the type once.

Example 4.6 (Global Type Reductions). We demonstrate two reduction paths for a global type

$$G = \mathbf{p} \to \mathbf{q} : Hello(x : \mathrm{int}\{x < 0\}).\mathbf{r} \to \mathbf{s} : Hola(y : \mathrm{int}\{y > x\}).\mathsf{end}.$$

Note that the two messages are not causally related (they have disjoint subjects). We have the following two reduction paths of $\langle \varnothing \prec G\rangle$ (omitting payload in LTS actions):

$$\langle \varnothing \prec G\rangle$$
$$[\text{G-Pfx}] \xrightarrow{\mathbf{p}\to\mathbf{q}:Hello} \langle x^{\{\mathbf{p},\,\mathbf{q}\}} : \mathrm{int}\{x < 0\} \prec \mathbf{r} \to \mathbf{s} : Hola(y : \mathrm{int}\{y > x\}).\mathsf{end}\rangle$$
$$[\text{G-Pfx}] \xrightarrow{\mathbf{r}\to\mathbf{s}:Hola} \langle x^{\{\mathbf{p},\,\mathbf{q}\}} : \mathrm{int}\{x < 0\}, y^{\{\mathbf{r},\,\mathbf{s}\}} : \mathrm{int}\{y > x\} \prec \mathsf{end}\rangle$$

$$\langle \varnothing \prec G\rangle$$
$$[\text{G-Cnt}] \xrightarrow{\mathbf{r}\to\mathbf{s}:Hola} \langle x^\varnothing : \mathrm{int}\{x < 0\}, y^{\{\mathbf{r},\,\mathbf{s}\}} : \mathrm{int}\{y > x\} \prec \mathbf{p} \to \mathbf{q} : Hello(x : \mathrm{int}\{x < 0\}).\mathsf{end}\rangle$$
$$[\text{G-Pfx}] \xrightarrow{\mathbf{p}\to\mathbf{q}:Hello} \langle x^{\{\mathbf{p},\,\mathbf{q}\}} : \mathrm{int}\{x < 0\}, y^{\{\mathbf{r},\,\mathbf{s}\}} : \mathrm{int}\{y > x\} \prec \mathsf{end}\rangle$$

Semantics of Local Types. We define the LTS semantics of local types in Fig. 14. Similar to global type LTS semantics, we include the local context Σ in the semantics. Therefore, the judgements of local LTS reductions have form $\langle \Sigma \prec L\rangle \xrightarrow{\alpha} \langle \Sigma' \prec L'\rangle$. When defining the LTS semantics, we also use judgements of form $\langle \Sigma \prec L\rangle \xrightarrow{\epsilon} \langle \Sigma' \prec L'\rangle$. It represents a silent action that can occur without an observed action. We write $\xrightarrow{\epsilon}{}^*$ to denote the reflexive transition closure of silent actions $\xrightarrow{\epsilon}$.

We first have a look at silent transitions. [E-Phi] allows the variable in a silent type to be added into the local context in the irrelevant form. This rule allows local roles to obtain knowledge from the messages in the global protocol without their participation.

[E-Cnt] allows prefixed local type to make a silent transition, if all of its continuations are allowed to make a silent transition to reach the same context. The rule allows a prefixed local type to obtain new knowledge about irrelevant variables, if such can be obtained in all possible continuations.

[E-Rec] unfolds recursive local types, analogous to the unfolding of global types.

For concrete transitions, we have [L-Send] (resp. [L-Recv]) to reduce a local type with a sending (resp. receiving) prefix, if the action label is in the set of labels in the local type. The resulting

$$[\text{E-Cnt}] \quad \frac{\dagger \in \{\&, \oplus\} \qquad \forall j \in I. \qquad \langle \Sigma \cup \{x_j^0 : T_j\} \prec L_j \rangle \xrightarrow{\epsilon} \langle \Sigma' \prec L_j' \rangle}{\langle \Sigma \prec \mathbf{q} \dagger \{l_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\epsilon} \langle \Sigma' \prec \mathbf{q} \dagger \{l_i(x_i : T_i).L_i'\}_{i \in I} \rangle}$$

$$[\text{E-Rec}] \quad \frac{}{\langle \Sigma \prec \mu\mathsf{t}\,(x : T)\langle x := E\rangle.L \rangle \xrightarrow{\epsilon} \langle \Sigma \cup \{x^\omega : T\} \prec L[\mu\mathsf{t}\,(x : T).L/\mathsf{t}] \rangle}$$

$$\boxed{\langle \Sigma \prec L \rangle \xrightarrow{\epsilon} \langle \Sigma' \prec L' \rangle} \qquad [\text{E-Phi}] \quad \frac{}{\langle \Sigma \prec l(x : T).L \rangle \xrightarrow{\epsilon} \langle \Sigma \cup \{x^0 : T\} \prec L \rangle}$$

$$[\text{L-Send}] \quad \frac{j \in I}{\langle \Sigma \prec \mathbf{q}\oplus\{l_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:l_j(x_j:T_j)} \langle \Sigma \cup \{x_j^\omega : T_j\} \prec L_j \rangle}$$

$$[\text{L-Recv}] \quad \frac{j \in I}{\langle \Sigma \prec \mathbf{p}\&\{l_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:l_j(x_j:T_j)} \langle \Sigma \cup \{x_j^\omega : T_j\} \prec L_j \rangle}$$

$$\boxed{\langle \Sigma \prec L \rangle \xrightarrow{\alpha} \langle \Sigma' \prec L' \rangle} \qquad [\text{L-Eps}] \quad \frac{\langle \Sigma \prec L \rangle \xrightarrow{\epsilon} \langle \Sigma'' \prec L'' \rangle \qquad \langle \Sigma'' \prec L'' \rangle \xrightarrow{\alpha} \langle \Sigma' \prec L' \rangle}{\langle \Sigma \prec L \rangle \xrightarrow{\alpha} \langle \Sigma' \prec L' \rangle}$$

Fig. 14. LTS Semantics for Local Types

context contains the variable in the message as a concrete variable, since the role knows the value via communication. The resulting local type is the continuation corresponding to the action label.

In addition, [L-Eps] permits any number of silent actions to be taken before a concrete action.

Remark 4.7 (Reductions for Empty Session Types). We consider empty session types to be reducible, since it is not possible to distinguish which types are inhabited. However, it does not invalidate the safety properties of endpoints, since no such endpoints can be implemented for an empty session type.

Relating Semantics of Global and Local Types. We extend the LTS semantics to a collection of local types in Definition 4.8, in order to prove that projection preserves semantics. We define the semantics in a synchronous fashion.

The set of local types reduces with an action $\alpha = \mathbf{p} \to \mathbf{q} : l(x : T)$, if the local type for role $\mathbf{p}$ and $\mathbf{q}$ both reduce with that action α. All other roles in the set of the local types are permitted to make silent actions (ϵ actions).

Our definition deviates from the standard definition [Deniélou and Yoshida 2013, Def. 3.3] in two ways: One is that we use a synchronous semantics, so that one action involves two reductions, namely at the sending and receiving sides. Second is that we use contexts and silent transitions in the LTS semantics. The original definition requires all non-action roles to be identical, whereas we relax the requirement to allow silent transitions.

Definition 4.8 (LTS over a collection of local types). A configuration $s = \{\langle \Sigma_\mathbf{r} \prec L_\mathbf{r}\rangle\}_{\mathbf{r}\in\mathbb{P}}$ is a collection of local types and contexts, indexable via participants.

Let $\mathbf{p} \in \mathbb{P}$ and $\mathbf{q} \in \mathbb{P}$. We say $s = \{\langle \Sigma_\mathbf{r} \prec L_\mathbf{r}\rangle\}_{\mathbf{r}\in\mathbb{P}} \xrightarrow{\alpha=\mathbf{p}\to\mathbf{q}:l(x:T)} s' = \{\langle \Sigma_\mathbf{r}' \prec L_\mathbf{r}'\rangle\}_{\mathbf{r}\in\mathbb{P}}$ if

(1) $\langle \Sigma_\mathbf{p} \prec L_\mathbf{p}\rangle \xrightarrow{\alpha} \langle \Sigma_\mathbf{p}' \prec L_\mathbf{p}'\rangle$ and, $\langle \Sigma_\mathbf{q} \prec L_\mathbf{q}\rangle \xrightarrow{\alpha} \langle \Sigma_\mathbf{q}' \prec L_\mathbf{q}'\rangle$ and,

(2) for all $\mathbf{s} \in \mathbb{P}, \mathbf{s} \neq \mathbf{p}, \mathbf{s} \neq \mathbf{q}$. $\langle \Sigma_\mathbf{s} \prec L_\mathbf{s}\rangle \xrightarrow{\epsilon}^* \langle \Sigma_\mathbf{s}' \prec L_\mathbf{s}'\rangle$

For a closed global type G under context Γ, we show that the global type makes the same trace of reductions as the collection of local types obtained from projection. We prove it in Theorem 4.10.

Definition 4.9 (Association of Global Types and Configurations). Let $\langle \Gamma \prec G \rangle$ be a global context. The collection of local contexts *associated* to $\langle \Gamma \prec G \rangle$, is defined as the configuration $\{\langle \Gamma \prec G \rangle \restriction \mathbf{r}\}_{\mathbf{r} \in G}$. We write $s \Leftrightarrow \langle \Gamma \prec G \rangle$ if a configuration s is the associated to $\langle \Gamma \prec G \rangle$.

THEOREM 4.10 (TRACE EQUIVALENCE). *Let $\langle \Gamma \prec G \rangle$ be a closed global context and $s \Leftrightarrow \langle \Gamma \prec G \rangle$ be a configuration associated with the global context.*

$\langle \Gamma \prec G \rangle \xrightarrow{\alpha} \langle \Gamma' \prec G' \rangle$ *if and only if $s \xrightarrow{\alpha} s'$, where $s' \Leftrightarrow \langle \Gamma' \prec G' \rangle$.*

The theorem states that semantics are preserved after projection. Practically, we can implement local processes separately, and run them in parallel with preserved semantics.

We also show that a well-formed global type G has progress. This means that a well-formed global type does not get *stuck*, which implies deadlock freedom.

Definition 4.11 (Well-formed Global Types). A global type under typing context $\langle \Gamma \prec G \rangle$ is well-formed, if (1) G does not contain free type variables, (2) G is contractive [Pierce 2002, §21], and (3) for all roles in the protocol $\mathbf{r} \in G$, the projection $\langle \Gamma \prec G \rangle \restriction \mathbf{r}$ is defined.

We also say a global type G is well-formed, if $\langle \varnothing \prec G \rangle$ is well-formed.

THEOREM 4.12 (PRESERVATION OF WELL-FORMEDNESS). *If $\langle \Gamma \prec G \rangle$ is a well-formed global type under typing context, and $\langle \Gamma \prec G \rangle \xrightarrow{\alpha} \langle \Gamma' \prec G' \rangle$, then $\langle \Gamma' \prec G' \rangle$ is well-formed.*

Definition 4.13 (Progress). A configuration s satisfies progress, if either (1) For all participants $\mathbf{p} \in s$, $L_{\mathbf{p}} = \mathsf{end}$, or (2) there exists an action α and a configuration s' such that $s \xrightarrow{\alpha} s'$.

A global type under typing context $\langle \Gamma \prec G \rangle$ satisfies progress, if its associated configuration $s \Leftrightarrow \langle \Gamma \prec G \rangle$, exists and satisfies progress.

We also say a global type G satisfies progress, if $\langle \varnothing \prec G \rangle$ satisfies progress.

THEOREM 4.14 (PROGRESS). *If $\langle \Gamma \prec G \rangle$ is a well-formed global type under typing context, then $\langle \Gamma \prec G \rangle$ satisfies progress.*

THEOREM 4.15 (TYPE SAFETY). *If G is a well-formed global type, then for any global type under typing context $\langle \Gamma' \prec G' \rangle$ such that $\langle \varnothing \prec G \rangle \longrightarrow {}^*\langle \Gamma' \prec G' \rangle$, $\langle \Gamma' \prec G' \rangle$ satisfies progress.*

PROOF. Direct consequence of Theorem 4.12 and Theorem 4.14. □

5 EVALUATION

We evaluate the expressiveness and performance of our toolchain SESSION*. We describe the methodology and setup (§ 5.1), and comment on the compilation time (§ 5.2) and the execution time (§ 5.3). We demonstrate the expressiveness of SESSION* (§ 5.4) by implementing examples from the session type literature and comparing with related work. The source files of the benchmarks used in this section are included in our artifact, along with a script to reproduce the results.

5.1 Methodology and Setup

We measure the time to generate the CFSM representation from a SCRIBBLE protocol (*CFSM*), and the time to generate F* code from the CFSM representation (*F* APIs*). Since the generated APIs in F* need to be type-checked before use, we also measure the type-checking time for the generated code (*Gen. Code*). Finally, we provide a simple implementation of the callbacks and measure the type-checking time for the callbacks against the generated type (*Callbacks*).

```
global protocol PingPong_n(role A, role B) {
  choice at A { Ping(x_1:int) from A to B;                    Pong(y_1:int) from B to A; @"y_1>x_1"
                Ping(x_2:int) from A to B; @"x_2>y_1"  Pong(y_2:int) from B to A; @"y_2>x_2"
                ...
                Ping(x_n:int) from A to B; @"x_n>y_{n-1}" Pong(y_n:int) from B to A; @"y_n>x_n"
                do PingPong_n(A, B); }
          or { Bye() from A to B;                             Bye() from B to A; } }
```

Fig. 15. Ping Pong Protocol (Parameterised by Protocol Length n)

To execute the protocols, we need a network transport to connect the participants by providing appropriate sending and receiving primitives. In our experiment setup, we use the standard library module `FStar.Tcp` to establish TCP connections between participants, and provide a simple serialisation module for base types. Due to the small size of our payloads, we set `TCP_NODELAY` to avoid the delays introduced by the congestion control algorithms. Since our entry point to execute the protocol is parameterised by the connection/transport type, the implementation may use other connections if developers wish, e.g. an in-memory queue for local setups. We measure the execution time of the protocol (*Execution Time*).

To measure the overhead of our implementation, we compare against an implementation of the protocol without session types or refinement types, which we call *bare implementation*. In this implementation, we use the same sending and receiving primitives (i.e. `connection`) as in the toolchain implementation. The bare implementation is in a series of direct calls of sending and receiving primitives, for the same communication pattern, but without the generated APIs.

We use a Ping Pong protocol (Fig. 15), parameterised by the protocol length, i.e. the number of Ping Pong messages n in a protocol iteration. When the protocol length n increases, the number of CFSM states increases linearly, which gives rise to longer generated code and larger generated types. In each Ping Pong message, we include payload of increasing numbers, and encode the constraints as protocol refinements.

We study its effect on the compilation time (§ 5.2) and the execution time (§ 5.3). We run the experiment on varying sizes of n, up to 25. Larger sizes of n leads to unreasonably large resource usage during type-checking in $F^\star$. Table 1 reports the results for the Ping Pong protocol in Fig. 15.

We run the experiments under a network of latency of 0.340ms (64 bytes ping), and repeat each experiment 30 times. Measurements are taken using a machine with Intel i7-7700K CPU (4.20 GHz, 4 cores, 8 threads), 16 GiB RAM, operating system Ubuntu 18.04, OCaml compiler version 4.08.1, $F^\star$ compiler commit 8040e34a, Z3 version 4.8.5.

5.2 Compilation Time

CFSM and $F^\star$ Generation Time. We measure the time taken for Scribble to generate the CFSM from the protocol in Fig. 15, and for the code generation tool to convert the CFSM to $F^\star$ APIs. We observe from Table 1 that the generation time for CFSMs and $F^\star$ APIs is short. It takes less than 1 second to complete the generation phase for each case.

Type-checking Time of Generated Code and Callbacks. We measure the time taken for the generated APIs to type-check in $F^\star$. We provide a simple $F^\star$ implementation of the callbacks following the generated APIs, and measure the time taken to type-check the callbacks.

The increase of type-checking time is non-linear with regard to the protocol length. We encode CFSM states as records corresponding to local typing contexts. In this case, the size of local typing contexts and the number of type definitions grows linearly, giving rise to a non-linear increase. Moreover, the entry point function is likely to cause non-linear increases in the type-checking time.

Table 1. Time Measurements for Ping Pong Protocol

| Protocol | Generation Time | | Type Checking Time | | Execution Time |
Length (n)	CFSM	F* APIs	Gen. Code	Callbacks	(100 000 ping-pongs)
bare	n/a	n/a	n/a	n/a	28.79s
1	0.38s	0.01s	1.28s	0.34s	28.75s
5	0.48s	0.01s	3.81s	1.12s	28.82s
10	0.55s	0.01s	14.83s	1.34s	28.84s
15	0.61s	0.01s	42.78s	1.78s	n/a
20	0.69s	0.02s	98.35s	2.54s	28.81s
25	0.78s	0.02s	206.82s	3.87s	28.76s

The long type-checking time of the generated code could be avoided if the developer chooses to trust our toolchain to always generate well-typed F* code for the entry point. The entry point would be available in an *interface file* (cf. OCaml .mli files), with the actual implementation in OCaml instead of F*[5]. There would otherwise be no changes in the development workflow. Although neither does type-checking time of the callback implementation fit a linear pattern, it remains within reasonable time frame.

5.3   Runtime Performance (Execution Time)

We measure the execution time taken for an exchange of 100,000 ping pongs for the toolchain and bare implementation under the experiment network. The execution time is dominated by network communication, since there is little computation to be performed at each endpoint.

   We provide a bare implementation using a series of direct invocations of sending and receiving primitives, in a compatible way to communicate with generated APIs. The bare implementation does not involve a record of callbacks, which is anticipated to run faster, since the bare implementation involves fewer function pointers when calling callbacks. Moreover, the bare implementation does not construct *state records*, which record a backlog of the communication, as the protocol progresses. To measure the performance impact of book-keeping of callback and state records, we run the Ping Pong protocol from Fig. 15 for a protocol of increasing size (number of states and generated types), i.e. for increasing values of n. All implementations, including *bare* are run until 100,000 ping pong messages in total are exchanged[6].

   We summarise the results in Table 1. Despite the different protocol lengths, there are *no significant changes* in execution time. Since the execution is dominated by time spent on communication, the measurements are subject to network fluctuations, difficult to avoid during the experiments. We conclude that our implementation does not impose a large overhead on the execution time.

5.4   Expressiveness

We implement examples from the session type literature, and add refinements to encode data dependencies in the protocols. We measure the time taken for code generation and type-checking, and present them in Table 2. The time taken in the toolchain for examples in the session type literature is usually short, yet we demonstrate that we are able to implement the examples easily with our callback style API. Moreover, the time taken is incurred at the compilation stage, hence there is no overhead for checking refinements by our runtime.

---

[5]Defining a signature in an interface file, and providing an implementation in the target language (OCaml) allows the F* compiler to *assume* the implementation is correct. This technique is used frequently in the standard library of F*. This is not to be confused with implementing the endpoints in OCaml instead of F*, as that would bypass the F* type-checking.
[6]For $n = 1$, we run 100,000 iterations of recursion; for $n = 10$, we run 10,000 iterations, etc. Total number of ping pong messages exchanged by two parties remain the same.

Table 2. Selected Examples from Literature

Example (Endpoint)	Gen. / TC. Time	MP	RV	IV	STP
Two Buyer [a] (A)	0.46s / 2.33s	✓	✗	✓	✓[†]
Negotiation [b] (C)	0.46s / 1.59s	✗	✓	✗	✗
Fibonacci [c] (A)	0.44s / 1.58s	✗	✓	✗	✗
Travel Agency [d] (C)	0.62s / 2.36s	✓	✗	✗	✓[†]
Calculator [c] (C)	0.51s / 2.30s	✗	✗	✗	✓
SH [e] (P)	1.16s / 4.31s	✓	✗	✓	✓[†]
Online Wallet [f] (C)	0.62s / 2.67s	✓	✓	✗	✗
Ticket [g] (C)	0.45s / 1.90s	✗	✓	✗	✗
HTTP [h] (S)	0.55s / 1.79s	✗	✗	✗	✓[†]

MP	Multiparty Protocol
RV	Uses Recursion Variables
IV	Irrelevant Variables
STP	Implementable in STP
✓[†]	STP requires *dynamic* checks
[a]	[Honda et al. 2016]
[b]	[Demangeon and Honda 2012]
[c]	[Hu and Yoshida 2016]
[d]	[Hu et al. 2008]
[e]	[Neykova et al. 2018]
[f]	[Neykova et al. 2013]
[g]	[Bocchi et al. 2013]
[h]	[Fielding and Reschke 2014]

We also compare the expressiveness of our work with two most closely related works, namely Bocchi et al. [2010] and Neykova et al. [2018], which study refinements in MPST (also see § 6). Neykova et al. [2018] (Session Type Provider, STP) implements limited version of refinements in the SCRIBBLE toolchain. Our version is strictly more expressive than STP for two reasons: (1) support for recursive variables to express invariants and (2) support for irrelevant variables. Fig. 16 illustrates those features and Table 2 identifies which of the implemented examples use them.

```
protocol Adder(role S, role C)          protocol Broadcast(role A, role B, role C)
@"S[acc:=0]" {                          {
  Num(x:int)    from C to S; @"x≥0"       Broadcast(x:int) from A to B; @"x≥0"
  Sum(sum:int) from S to C; @"sum=acc+x"  // C does not learn y≥0 in STP
  do Adder(S, C); @"S[sum]" }             Broadcast(y:int) from A to C; @"x=y" }
  (a) Accumulator (using Recursive Invariants)   (b) Broadcasting (using Irrelevant Variables)
```

Fig. 16. Example Protocols Demonstrating Additional Expressiveness to [Neykova et al. 2018]

In STP, when recursion occurs, all information about the variables is lost at the end of an iteration, hence their tool does not support even the simple example in Fig. 16a. In contrast, our work retains the recursion variables, which are available throughout the recursion. Additionally, the endpoint projection in STP is more conservative with regards to refinements. Whilst there must be no variables unknown to a role in the refinements attached to a message for the sending role, there may be unknown variables to the receiving role. The part unknown to the receiving role is discarded (hence weakening the pre-condition). In our work such information can still be retained and used for type checking, thanks to irrelevant variables.

In Bocchi et al. [2010], a global protocol with assertions must be *well-asserted* (§3.1). In particular, the *history sensitivity* requirement states: *"A predicate guaranteed by a participant **p** can only contain those interaction variables that **p** knows."* Our theory lifts this restriction by allowing variables unknown to a sending role to be used in the global or local type, whereas such variables cannot be used in the implementation. For example, Example 4.1 fails the well-asserted requirement in [Bocchi et al. 2010]. In the refinement $x = z$ for variable z (for message label *Trd*), the variable x is not known to **C**, hence the protocol would not be well-asserted. In our setup, such protocol is permitted, the endpoint implementation for **C** can provide the value y received from **B** to satisfy the refinement type — The SMT solver can validate the refinement from the transitivity of equality.

6 RELATED WORK

We summarise the most closely related works in the areas of refinement and session types. For a detailed survey on theory and implementations of session types, see Gay and Ravara [2017].

Refinement Types for Verification and Reasoning. Refinement types were introduced to allow recursive data structures to be specified in more details using predicates [Freeman and Pfenning 1991]. Subsequent works on the topic [Bengtson et al. 2011; Schmid and Kuncak 2016; Vazou et al. 2014, 2017] utilise SMT solvers, such as Z3 [De Moura and Bjørner 2008], to aid the type system to decide a semantic subtyping relation [Bierman et al. 2012] using SMT encodings. Refinement types have been applied to numerous domains, such as resource usage analysis [Handley et al. 2019; Knoth et al. 2020], secure implementations [Bengtson et al. 2011; Bhargavan et al. 2010], information control flow enforcements [Polikarpova et al. 2020], and theorem proving [Vazou et al. 2017]. Our aim is to utilise refinement types for the specification and verification of distributed protocols, by combining refinement and session types in a single practical framework.

Implementation of Session Types. Neykova et al. [2018] provides an implementation of MPST with assertions using Scribble and F#. Their implementation, the session type provider (STP), relies on code generation of fluent (class-based) APIs, initially described in [Hu and Yoshida 2016]. Each protocol state is implemented as a class, with methods corresponding to the possible transitions from that state. It forces a programming style that not only relies extensively on method chaining, but also requires dynamic checks to ensure the linearity of channel usage. Our work differs from STP in multiple ways. First, we extend the Scribble toolchain to support *recursion variables*, allowing refinements on recursions, hence improving expressiveness. In this way, developers can specify dependencies across recursive calls, which is not supported in STP. Second, we depart from the class-based API generation, and generate a callback-based API. Our approach has the advantage that the linear usage of channels is ensured by construction, saving dynamic checks for channels. Third, we use refinement types in F⋆ to verify refinements statically, in contrast, STP performs dynamic evaluations to validate assertions in protocols. Finally, the metatheory of session types extended with refinements was not developed in their work.

Several other MPST works follow a similar technique of class-based API generation to overcome limitations of the type system in the target language, e.g. Castro et al. [2019] for Go, Ng et al. [2015] for C. All of the above works, suffer from the same limitations – they detect linearity violations at runtime, and offer no static alternative. Indeed, to our knowledge, Imai et al. [2020] provide the only MPST implementation which *statically* checks linearity violation. It relies on specific type-level OCaml features, and a monadic programming style. Our work proposes generation of a callback-styled API from MPST protocols. Although our target language is F⋆, the callback-styled API code generation technique is applicable to any mainstream programming language.

Dependent and Refinement Session Types. Bocchi et al. [2010] propose a multiparty session π-calculus with logical assertions. By contrast, our formulation of RMPST is based on refinement types, projection with silent prefixes and correspondence with CFSMs, to target practical code generation, such as for F⋆. They do not formulate any semantics for global types nor prove an equivalence between refined global types and projections, as in this paper. Toninho and Yoshida [2017] extend MPST with value dependent types. Invariants on values are witnessed by proof objects, which then may be erased at runtime. Our work uses refinement types, which follows the principle naturally, since refinements that appear in types are proof-irrelevant and can be erased safely. These works are limited to theory, whereas we provide an implementation.

De Muijnck-Hughes et al. [2019] propose an Embedded Domain Specific Language (EDSL) approach of implementing multiparty sessions (analogous to MPST) in Idris. They use value dependent types in Idris to define combinators, with options to specify data dependencies, contrary to our approach of code generation. However, the combinators only describe the sessions, and how to implement and execute the sessions remains unanswered. Our work provides a complete toolchain from protocol description to implementation and verification.

In the setting of binary session types, Das and Pfenning [2020] extend session types with arithmetic refinements, with application to work analysis for computing upper bounds of work from a given session type. Thiemann and Vasconcelos [2019] extend binary session types with label dependent types. In the setup of their work, specification of arithmetic properties involves complicated definitions of inductive arithmetic relations and functions. In contrast, we use SMT solvers, which have built-in functions and relations for arithmetic. Furthermore, there is no need to construct proofs manually, since SMT solvers find the proof automatically, which enhances usability and ergonomics. Hinrichsen et al. [2019] combine binary session types with concurrent separation logic, allowing reasoning about mixed-paradigm concurrent programs, and planned to extend the framework to MPST. Swamy et al. [2020] provide a framework of concurrent separation logic in F*, and demonstrate its expressiveness by showing how (dependent) binary session types can be represented in the logic and used in reasoning. Our work is based on the MPST theory, subsuming binary session types. We also implement a toolchain for developers to use.

Bhargavan et al. [2009] use refinement types to implement a limited form of multiparty session types. Session types are encoded in refinement types via code generation. The specification language they use, albeit similar to MPST, has limited expressive power. Only patterns of interactions where participants alternate between sending and receiving are permitted. Moreover, they do not study data dependencies in protocols, hence they can neither specify, nor verify constraints on payloads or recursions. We use refinement types to specify constraints and dependencies in multiparty protocols, and use the F* compiler [Swamy et al. 2016] for verifying the endpoint implementations. The verified endpoint program does not only comply to the multiparty protocol, enjoying the guarantees provided by the original MPST theory (deadlock freedom, session fidelity), but also satisfies additional guarantees provided by refinement types with respect to data constraints.

7 CONCLUSIONS AND FUTURE WORK

We present a novel toolchain for implementing refined multiparty session types (RMPST), which enables developers to use Scribble, a protocol description language for multiparty session types, and F*, a state-of-the-art verification-oriented programming language, to implement a multiparty protocol and statically verify endpoint implementations. To the best of the authors' knowledge, this is the first work on *statically* verified multiparty protocols with *refinement* types. We extend the theory of multiparty session types with data refinements, and present a toolchain that enables developers to *specify* multiparty protocols with data dependencies, and *implement* the endpoints using generated APIs in F*. We leverage the advanced typing system in F* to encode local session types for endpoints, and validate the data dependencies in the protocol statically.

The verified endpoint program in F* is extracted into OCaml, where the refinements are *erased* — adding *no runtime overhead* for refinements. The callback-styled API avoids linearity checks of channel usage by internalising communications in generated code. We evaluate our toolchain and demonstrate that our overhead is small compared to an implementation without session types.

Whereas refinement types express the data dependencies of multiparty protocols, the availability of refinement types in general purpose mainstream programming languages is limited. For future work, we wish to study how to mix participants with refined implementation and those without, possibly using a gradual typing system [Igarashi et al. 2019; Lehmann and Tanter 2017].

REFERENCES

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* Volume 8, Issue 1 (March 2012). https://doi.org/10.2168/LMCS-8(1:29)2012

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2–3 (July 2016), 95–230. https://doi.org/10.1561/2500000031

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 8.

Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. 2009. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *2009 22nd IEEE Computer Security Foundations Symposium*. 124–140. https://doi.org/10.1109/CSF.2009.26

Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2010. Modular Verification of Security Protocol Code by Typing. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 445–456. https://doi.org/10.1145/1706299.1706350

Gavin M Bierman, Andrew D Gordon, Cătălin Hriţcu, and David Langworthy. 2012. Semantic subtyping with an SMT solver. *Journal of Functional Programming* 22, 1 (2012), 31–105. https://doi.org/10.1017/S0956796812000032

Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. 2013. A Multiparty Multi-session Logic. In *Trustworthy Global Computing*, Catuscia Palamidessi and Mark D. Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–111.

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 162–176.

Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (April 1983), 323–342. https://doi.org/10.1145/322374.322380

David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290342

Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:18. https://doi.org/10.4230/LIPIcs.CONCUR.2020.13

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. 2019. Value-Dependent Session Design in a Dependently Typed Language. In Proceedings *Programming Language Approaches to Concurrency- and Communication-cEntric Software*, Prague, Czech Republic, 7th April 2019 *(Electronic Proceedings in Theoretical Computer Science, Vol. 291)*, Francisco Martins and Dominic Orchard (Eds.). Open Publishing Association, 47–59. https://doi.org/10.4204/EPTCS.291.5

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7454)*, Maciej Koutny and Irek Ulidowski (Eds.). Springer, 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

Pierre-Malo Denielou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* Volume 8, Issue 4 (Oct. 2012). https://doi.org/10.2168/LMCS-8(4:6)2012

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming (LNCS, Vol. 7966)*. Springer, Berlin, Heidelberg, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

Roy T. (Ed.) Fielding and Julian F. (Ed.) Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor. 1–89 pages. https://www.rfc-editor.org/info/rfc7230

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

Simon Gay and António Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools*. River Publishers. https://doi.org/10.13052/rp-9788793519817

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL, Article 24 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371092

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2019. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 6 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371074

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63 (2016), 1–67. Issue 1-9. https://doi.org/10.1145/2827695

Raymond Hu. 2017. Distributed Programming Using Java APIs Generated from Session Types. *Behavioural Types: from Theory to Tools* (2017), 287–308.

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering (LNCS, Vol. 9633)*. Springer, Berlin, Heidelberg, 401–418. https://doi.org/10.1007/978-3-662-49665-7_24

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5142)*, Jan Vitek (Ed.). Springer, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. 2019. Gradual session types. *Journal of Functional Programming* 29 (2019), e17. https://doi.org/10.1017/S0956796819000169

Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming with Global Protocol Combinators. https://github.com/keigoi/ocaml-mpst *(LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. To appear in ECOOP'20.

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-ocaml: A session-based library with polarities and lenses. *Science of Computer Programming* 172 (2019), 135 – 159. https://doi.org/10.1016/j.scico.2018.08.005

Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid Resource Types. *Proc. ACM Program. Lang.* 4, ICFP, Article 106 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408988

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/3009837.3009856

Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-time API Generation of Distributed Protocols with Refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/3178372.3179495

Rumyana Neykova and Nobuko Yoshida. 2019. *Featherweight Scribble*. Springer International Publishing, Cham, 236–259. https://doi.org/10.1007/978-3-030-21485-2_14

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *Runtime Verification*, Axel Legay and Saddek Bensalem (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 358–363.

Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. 2015. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction (LNCS, Vol. 9031)*. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11

Dominic Orchard and Nobuko Yoshida. 2017. Session Types with Linearity in Haskell. *Behavioural Types: from Theory to Tools* (2017), 219–241.

Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 105 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408987

Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

Georg Stefan Schmid and Viktor Kuncak. 2016. SMT-based Checking of Predicate-qualified Types for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala* (Amsterdam, Netherlands) *(SCALA 2016)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2998392.2998398

Scribble Authors. 2015. Scribble: Describing Multi Party Protocols. http://www.scribble.org/. Accessed on 20th April 2020.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan
   Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-
   Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-
   SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, St. Petersburg, FL, USA, 256–270.
   https://doi.org/10.1145/2837614.2837655
Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore:
   An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4,
   ICFP, Article 121 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409003
Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.*, Article 67
   (Dec. 2019), 29 pages. https://doi.org/10.1145/3371135
Bernardo Toninho and Nobuko Yoshida. 2017. Certifying data in multiparty session types. *Journal of Logical and Algebraic
   Methods in Programming* 90 (2017), 61 – 83. https://doi.org/10.1016/j.jlamp.2016.11.005
Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell.
   In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden)
   (ICFP '14). ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161
Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala.
   2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017),
   31 pages. https://doi.org/10.1145/3158141
Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. *Statically Verified Refinements
   for Multiparty Protocols*. https://doi.org/10.5281/zenodo.3970760