Distributed Programming using Role-Parametric Session Types in Go

Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures

DAVID CASTRO, Imperial College London, United Kingdom
RAYMOND HU, Imperial College London, United Kingdom
SUNG-SHIK JONGMANS, Open University of the Netherlands, The Netherlands and Imperial College London, United Kingdom
NICHOLAS NG, Imperial College London, United Kingdom
NOBUKO YOSHIDA, Imperial College London, United Kingdom

This paper presents a framework for the static specification and safe programming of message passing protocols where the number and kinds of participants are *dynamically* instantiated.

We develop the first theory of *distributed* multiparty session types (MPST) to support parameterised protocols with indexed roles—our framework statically infers the different kinds of participants induced by a protocol definition as *role variants*, and produces *decoupled* endpoint projections of the protocol onto each variant. This enables safe MPST-based programming of the parameterised endpoints in distributed settings: each endpoint can be implemented separately by different programmers, using different techniques (or languages). We prove the decidability of role variant inference and well-formedness checking, and the correctness of projection.

We implement our theory as a toolchain for programming such role-parametric MPST protocols in Go. Our approach is to generate API families of lightweight, protocol- and variant-specific type wrappers for I/O. The APIs ensure a well-typed Go endpoint program (by native Go type checking) will perform only compliant I/O actions w.r.t. the source protocol. We leverage the abstractions of MPST to support the specification and implementation of Go applications involving multiple channels, possibly over mixed transports (e.g., Go channels, TCP), and channel passing via a unified programming interface. We evaluate the applicability and run-time performance of our generated APIs using microbenchmarks and real-world applications.

29

CCS Concepts: • **Computing methodologies → Distributed programming languages**; • **Software and its engineering → Source code generation**; **Concurrent programming languages**;

Additional Key Words and Phrases: multiparty session types, indexed roles, distributed programming, Go

ACM Reference Format:
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 (January 2019), 30 pages. https://doi.org/10.1145/3290342

Authors' addresses: David Castro, Department of Computing, Imperial College London, United Kingdom, d.castro-perez@imperial.ac.uk; Raymond Hu, Department of Computing, Imperial College London, United Kingdom, raymond.hu@imperial.ac.uk; Sung-Shik Jongmans, Department of Computer Science, Open University of the Netherlands, The Netherlands, ssj@ou.nl, Department of Computing, Imperial College London, United Kingdom; Nicholas Ng, Department of Computing, Imperial College London, United Kingdom, nickng@imperial.ac.uk; Nobuko Yoshida, Department of Computing, Imperial College London, United Kingdom, n.yoshida@imperial.ac.uk.

2475-1421/2019/1-ART29
https://doi.org/10.1145/3290342

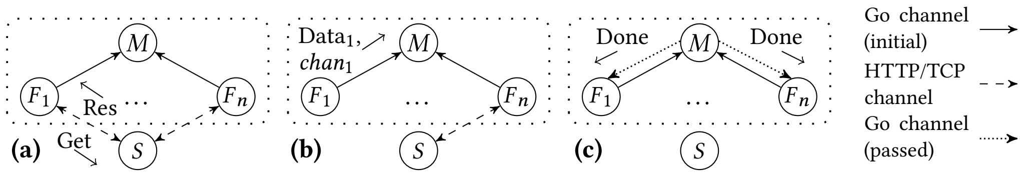


Fig. 1. Distributed and shared memory communications in a role-parametric parallel HTTP downloader.

1 BACKGROUND

1.1 Channel-Based Concurrent and Distributed Programming in Go

Go is a popular industrial systems language.[1] One of its primary design features is first-class language support for lightweight concurrency on multicore machines. Go offers easy spawning of parallel coroutines, called *goroutines*, that are transparently multiplexed over an underlying set of system threads. Goroutines communicate and synchronize via message passing over typed *channels*, designed to alleviate the difficulties of low-level mechanisms such as mutexes, condition variables and memory barriers commonly used in systems programming. As first-class objects, an interesting and *useful* feature is the ability to pass *channels over channels*.

Go is also well-established in distributed systems; e.g., it is the implementation language of frameworks such as Kubernetes, Docker and Jaeger. As the aforementioned concurrency features of Go are specific to shared memory, a significant class of distributed programming in Go is conducted using channel-based networking libraries via TCP, HTTP, etc. as transports. Developers appreciate Go since distributed programming in practice often involves local concurrency: goroutines and channels are effective for dealing locally with the inherent asynchrony of distributed interactions.

We illustrate such an application that integrates shared memory and distributed concurrency as a running example, a parallel downloader (e.g., HTTP) which we refer to as Pget.[2] Fig. 1 depicts the components of the application and the communication structures that arise.

- **(a)** There are three categories of participants, one Master (M), $n > 0$ Fetchers (F), and one Server (S). M creates a *worker pool* of n goroutines to serve as Fs, where the value of n is set at *run-time*, and shares a Go channel with each to retrieve the data. Each F performs its download task (by a Get/Res message exchange) with S concurrently via a separate HTTP channel.
- **(b)** When an F finishes its download, it passes to M the data and a *continuation channel* over the initially shared Go channel (this pattern is as in the implementation of `htcat`[2]).
- **(c)** The passed channel (dotted line) permits M to relay the next message *type* in the protocol after receiving a Data: e.g., to give the F another download task, or to end the goroutine (Done).

Go channels are homogeneously typed: the syntax of channel types is `chan T` for a given type `T`. Channel passing as above (i.e., bundling the continuation channel into the current message) is a way to affect the causality between the communications of *different* message types, as a safer alternative to declaring and allocating all channels upfront: passing the continuation channel as part of using the "current" channel helps prevent using them out of the intended order.

---

[1] https://golang.org/. Companies using Go: https://github.com/golang/go/wiki/GoUsers. Go success stories: https://github.com/golang/go/wiki/SuccessStories. Survey of 3,595 Go users: https://blog.golang.org/survey2016-results.

[2] Pget is based on `htcat` (https://github.com/htcat/htcat), a tool for parallelising HTTP `GET`s written in Go using channels (and channel passing) and the standard `net` package, with performance gains compared to `curl`.

1.2 Key Programming Challenges

The Pget example demonstrates some of the key challenges faced by distributed programmers in many engineering languages, including recent languages like Go. As a terminology, we shall refer to Go channels as *shared memory channels*, signifying intra-process message passing.

Communication and concurrency errors Go offers convenient primitives for shared memory channels, but does *not* offer any language support against classical errors such as *deadlocks* (goroutines stuck on mutually blocking inputs). In a recent survey,[1] users perceived this to be the main challenge in Go: "*We asked how strongly people agreed [with] various statements about Go. [...] Users* ***least agreed*** *that they are able to* ***effectively debug uses of Go's concurrency features***." One factor is that Go's channel *types* are limited. They do not *at heart* constrain the *direction* of communication; [3] nor reflect the *causality* of communications across *separate* channels, which also gives rise to *reception errors* (receiving an incorrect message type). These problems apply similarly to uses of distributed channel libraries, that often are effectively "untyped" in practice.

Disparate communication abstractions Key to understanding an application like Pget *as a whole* is the *choreography* of I/O behaviours by every participant across the multiple channels. At the *specification* level, there is first the question of how to statically specify protocols where the number and kinds of participants are dynamically determined: we refer to such protocols as having *dynamically-instantiated communication structures*. In practice many protocols are only informally specified, itself a cause of errors. This problem is compounded at the *implementation* level, where *disparate* primitives/libraries are used to implement heterogeneous parts of an endpoint (e.g., shared memory and HTTP in *F*)—even with an adequate specification, the programming abstractions do not guide a correct implementation of the *overall* application protocol nor facilitate its verification.

1.3 Multiparty Session Types: Motivations

Towards addressing these challenges, in this paper, we present a new, practical framework for the static specification and safe implementation of distributed Go programs, centred around a pivotal extension of the theory of *multiparty session types* (MPST) [Coppo et al. 2016; Honda et al. 2016]. Our general motivation for using MPST to address the challenges in § 1.2 is as follows.

In common practice, channel-oriented *communications* programming, embodied by standard networking libraries in many languages (including those with static *data* typing), is often effectively "untyped": for example, standard TCP socket APIs simply expose a raw byte stream in each communication direction. Higher-level and more recent facilities, such as service-oriented APIs and frameworks (e.g., SOAP, REST or Apache Thrift) and Go channels, can offer the improvement of *message-type safety*: the messages to be sent and received can be statically checked to be of known types. However, this still falls short of what is ultimately desired for communications-oriented programming in general: *protocol compliance*. The aforementioned facilities mask this limitation to certain extents: service-oriented frameworks essentially hardcode interaction *structures* to call-return patterns, thus reducing protocol compliance (for *individual* invocations) to message-type safety; Go channels are homogeneously typed, and often used with additional restrictions on the communication direction (via ad hoc casting of channel types).

The above limitations of current practices are readily exposed in many applications. For example, non-trivial service-based applications often involve, as a *whole*, the composition of multiple, smaller services: e.g., invoke service A *then* B, which in turn uses *either* C (then the protocol is *repeated* from the start) *or* D, and so on; such scenarios are increasingly promoted by architectures such as microservices that favour fine-grained service decomposition. In the setting of Go channels, such interaction structures require multiple *independent* channels to cater for the range of data types

[3] Go's directed channel types (`<-chan T` or `chan<- T`) are derived by ad hoc casting, and offer no guarantees against deadlock.

and communication directions. In contrast to the safety benefits of data typing enjoyed for "local" computations, programming of such communications suffers from errors arising from *protocol violations* (i.e., *non-protocol-compliant* I/O actions): despite message-type safety, these include the classical *reception errors* (receiving an out-of-order message, e.g., an incorrect invocation of B before A), *deadlocks* (a wait-for cycle of input dependencies) and *orphan messages* ("leftover" messages). The idea of MPST is to detect such errors at compile-time through static typing.

The rest of this paper summarises our contributions (§ 2), demonstrates our work through the running examples (§ 3), and presents our theory (§ 4), implementation (§ 5) and evaluation (§ 6). Our **Supplement**[4] gives additional examples and detailed proofs.

2 OUR CONTRIBUTIONS

2.1 In a Nutshell

(1) We develop the first theory of MPST to support role-parametric protocols in the traditional *distributed* spirit of MPST, including proofs of *decidability* (inferring "role variants"; checking well-formedness) and *correctness* of projection; § 2.2 details this contribution. Our theory is directly motivated by Go applications, but the foundations are *independent* of Go. Our approach thus also applies to other settings where shared-memory and distributed channel-based communication can be mixed (e.g., Rust).

(2) We implement our theory to give the first practical toolchain for MPST-based programming in Go. Our toolchain generates lightweight, *typed APIs* for users to implement the endpoint programs. Our toolchain is also the first to support practical *programming* of *role-parametric* MPST, targeting a language such as Go (cf., dependently typed session π-calculus). It ensures a statically well-typed endpoint program (i.e., by native Go type checking) will *never* perform a non-compliant I/O action w.r.t. to the run-time instantiation of the role-parametric protocol.

(3) Besides safety, we confer programmatic benefits of MPST to Go. Our toolchain enriches channel-over-channel passing in Go to *session delegation* (session-typed channel passing). Session code written using our generated APIs is also transport-*independent*: switching and mixing transports (e.g., Go channels, TCP) is safe and set by a single API argument.

(4) We demonstrate the applicability of our framework and run-time performance of our generated APIs by specifying and implementing a range of use cases from parallel algorithms and Internet applications, including modifying existing Go implementations of real-world applications—e.g., the overheads of our APIs are mostly negligible in programs adapted from [Gouy 2017].

We clarify the conditions for concrete applications of our practical framework:

- We target message passing applications where message delivery is *reliable* and *order-preserving* between each pair of participants in each direction (e.g, TCP, or FIFOs in shared memory). Our core theory is based on the standard *asynchronous* model of MPST, i.e., non-blocking outputs with blocking inputs, but our results also hold for synchronous communications.
- Our framework is top-down from a source protocol specification, which must be well-formed according to our definitions (§ 4). The expressiveness of our framework is attested by practical examples (§ 3), formal examples (§ 4.2), and a range of real-world applications (§ 6.2).

2.2 The Advances of this Paper to MPST

MPST basics. Multiparty session types (MPST) is one of the approaches in the field of behavioural type theory [Ancona et al. 2016; Hüttel et al. 2016] proposed to address the challenges discussed in § 1.2. Fig. 2 (a) depicts the standard top-down methodology of the originating MPST

[4] Technical report 2018/04, Department of Computing, Imperial College London. https://www.doc.ic.ac.uk/research/technicalreports/2018/#4

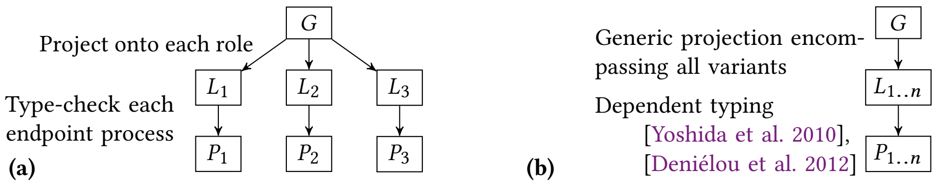


Fig. 2. Contrasting **(a)** the traditional top-down, distributed view of MPST [Coppo et al. 2016; Honda et al. 2016]; and **(b)** the "centralised" view of existing role-parametric MPST systems.

systems in the π-calculus [Coppo et al. 2016; Honda et al. 2016], which we illustrate by a small example: a ring communication structure between three Workers, W_1, W_2, and W_3.

$$G = \mathtt{W}_1 \longrightarrow \mathtt{W}_2 : \mathtt{T} \,.\, \mathtt{W}_2 \longrightarrow \mathtt{W}_3 : \mathtt{T} \,.\, \mathtt{W}_3 \longrightarrow \mathtt{W}_1 : \mathtt{T} \,.\, \mathtt{end}$$

G is a *global type*: a specification of the communication structure (i.e., *protocol*) between the participants (abstracted as *roles*) from a global perspective. G says W_1 first sends a $\mathtt{T}$ message to W_2, who then sends a message to W_3, who finally sends a message to W_1. For each role r, the global type is then *projected* to a *local type*, that describes the localised I/O actions from r's perspective:

$$L_1 = \mathtt{W}_2 \,!\, \mathtt{T} \,.\, \mathtt{W}_3 \,?\, \mathtt{T} \,.\, \mathtt{end} \qquad L_2 = \mathtt{W}_1 \,?\, \mathtt{T} \,.\, \mathtt{W}_3 \,!\, \mathtt{T} \,.\, \mathtt{end} \qquad L_3 = \mathtt{W}_2 \,?\, \mathtt{T} \,.\, \mathtt{W}_1 \,!\, \mathtt{T} \,.\, \mathtt{end}$$

L_1 says W_1 should first send (!) a $\mathtt{T}$ message to W_2, *followed by* receiving (?) a $\mathtt{T}$ message from W_3; the $\mathtt{W}_2 \longrightarrow \mathtt{W}_3$ interaction is transparent to W_1. Local types are used to statically *type check* endpoint programs (formally, session π-calculus processes) implementing these roles: intuitively, the typing checks protocol compliance by matching the structure of the I/O actions in the local type to a correspondingly structured usage of I/O primitives in the program. A well-typed system of processes, one for each role, is guaranteed free from reception errors and deadlocks.

A crucial design point of MPST is that projection promotes *modularity*: it decouples the programming (and verification) of each endpoint. This is especially important for distributed programming, which in addition to inter-process communications, may also be characterised by endpoints being *separately* implemented by different programmers, using different techniques (e.g., multithreaded, event-driven, etc.), technologies (e.g., client vs. server), and languages.

Addressing an open problem. One of the biggest challenges in MPST is *expressiveness*: essentially, to attain the strong static guarantees that MPST aims to provide, global types are syntactically limited and subject to conservative *well-formedness* and *projectibility* constraints (i.e., projection is a partial operator).

A crucial practical limitation of MPST concerns the lack of support for *role-parameterisation*, i.e., global and local types where roles are parameterised by indices. For instance, it should be possible to write a single global type for a ring communication structure of *any* size, instantiated dynamically; other applications include those involving parameterised worker/service instantiations (e.g., Pget), and many parallel algorithms. The original theory of MPST does not support such role-parameterisation, and while attempts have been made to extend the theory, these extensions ultimately had to sacrifice (1) general decidability of type checking and (2) modularity of projection.

This paper presents a new theory that is the first to support role-parametricity in MPST without the previous compromises, maintaining both decidability and modularity. Due to our new theory, we are able to contribute the first practical toolchain for role-parametric, distributed, MPST-based programming in an engineering language such as Go *without* relying on dependent types at the implementation level. Our framework guarantees only I/O actions that are compliant with the run-time instantiation of the role-parametric protocol are performed.

Comparison. To further clarify our contributions, we illustrate the approach of Deniélou et al. [2012]; Yoshida et al. [2010], the initial theoretical works that formulate a dependently typed MPST for protocols with indexed roles by adding a primitive recursion operator $\mathbf{R}$ to types and processes. The generalisation of the above example to a ring between $k \geq 2$ participants can be written as:

$$G = \Pi \mathtt{k}{:}I.(\mathbf{R}\, G'\, \lambda\mathtt{i}.\lambda\mathbf{x}.G'') \qquad G' = \mathtt{W[k]} \to \mathtt{W[0]} : \mathtt{T}\,.\,\mathtt{end} \qquad G'' = \mathtt{W[k-i-1]} \to \mathtt{W[k-i]} : \mathtt{T}\,.\,\mathbf{x}$$

where I is the parameter domain (≥2), `i` is an index variable, and $\mathbf{x}$ is a recursion variable. The use of $\mathbf{R}$ in G can essentially be read as: repeat G'' for `i` from `k-1` to `0`, then finish by doing G'.

In contrast to standard MPST, however, Fig. 2 (b) shows a corresponding top-down view of the methodology promoted by these works. G is projected to a *single* local type (called the *generic projection*) that encompasses the entire range of different index-value dependent behaviours *as one*.

$$L_{1..n} = \mathbf{R}\,(\mathtt{if}\ p = \mathtt{W[k]}\ (\mathtt{W[0]}\ !\ \mathtt{T}\,.\,\mathtt{end})\ \mathtt{else\,if}\ p = \mathtt{W[0]}\ (\mathtt{W[k]}\ ?\ \mathtt{T}\,.\,\mathtt{end})\ \mathtt{else\ end})$$
$$(\lambda\mathtt{i}.\lambda\mathbf{x}.\mathtt{if}\ p = \mathtt{W[k-i-1]}\ (\mathtt{W[k-i]}\ !\ \mathtt{T}\,.\,\mathbf{x})\ \mathtt{else\,if}\ p = \mathtt{W[k-i]}\ (\mathtt{W[k-i-1]}\ ?\ \mathtt{T}\,.\,\mathbf{x})\ \mathtt{else}\ \mathbf{x})$$

As the $\mathbf{R}$ operator iterates through the index range `k..0` for each participant p, the embedded index expression cases will spell out the three *distinct* behaviours present in the ring: those of `W[0]`, `W[1..k-1]`, and `W[k]`. We note that supplying the (valid) index domain, i.e., $\mathtt{k} \geq 2$, in their system *fixes* the type family—the intuitive case of a *two*-party ring requires declaring a separate type family (cf., $\mathtt{k} = 1$ is invalid in the above). Fixing the (finite) domain is required for decidability of type checking.

We now give the same example in our framework. The global type is:

$$G_{\text{Ring}} = \mathtt{W} \overset{1}{\Rightarrow} \mathtt{[1..k]} : \mathtt{T}\,.\,\mathtt{W[k]} \to \mathtt{W[1]} : \mathtt{T}\,.\,\mathtt{end}$$

where $\overset{1}{\Rightarrow}$ denotes a parameterised pipeline structure along the specified interval, i.e., $\mathtt{W[1]} \to \mathtt{W[2]} \ldots \mathtt{W[k-1]} \to \mathtt{W[k]}$; it is syntactic sugar (§ 4.2) for an instance of our *MPST-oriented* `foreach` construct: $\mathtt{foreach\ W\{i_1{:}1..k-1, i_2{:}2..k\}\ do\ W[i_1]} \to \mathtt{W[i_2]} : \mathtt{T}\,.\,\mathtt{cont}$ (cf. the generic $\mathbf{R}$). Our toolchain statically determines there are three *variants* of W, with *decoupled* projections:

$$L^{\mathtt{W[1]}}_{\text{Ring}} = \mathtt{W[2]\ !\ .\ W[k]\ ?} \qquad L^{\mathtt{W[2..k-1]}}_{\text{Ring}} = \mathtt{W[self-1]\ ?\ .\ W[self+1]\ !} \qquad L^{\mathtt{W[k]}}_{\text{Ring}} = \mathtt{W[k-1]\ ?\ .\ W[1]\ !}$$

(We omit the `T` *message labels* and `end`.) `self` denotes the run-time value of the local process identifier. From this single specification, the toolchain also determines the two valid endpoint *families*: that comprising variants $L^{\mathtt{W[1]}}_{\text{Ring}}$ and $L^{\mathtt{W[k]}}_{\text{Ring}}$ (when $\mathtt{k} = 2$), and when all three are involved ($\mathtt{k} > 2$).

3 METHODOLOGY OVERVIEW

3.1 Go Basics

We first summarise some basic Go features needed to understand our approach and code examples.

Types and variables. The following is a *type declaration* for a *defined type* (left), a *variable declaration* (centre), and a *shortened* declaration (right):

```
type Init struct { Err error; id uint64; Ept *S_1to1 }        var data Data        proto := Pget.New()
```

The left side defines a *struct type* named `Init`, that is a typed record with fields `Err` of type `error`, `id` of `uint64` and `Ept` of type `*S_1to1` (i.e., a *pointer type* with *base type* `S_1to1`). The declaration in the centre creates a variable `data` of type `Data`, automatically initialised to the *zero value* of that type (e.g., `nil` for interfaces and pointers). The right side is a shortened declaration for variable `proto` whose type and initial value is given by the expression `Pget.New()`.

Methods and interfaces. A method is a function with a *receiver*, i.e., a value upon which the method is invoked. The following is a method declaration (left) and a method call (right):

```
func (c *Foo) Job(a []Job) *M_3 { /* Method body omitted */ }                y := x.Job(myJobs)
```

The left side declares a method `Job`, with *receiver type* `*Foo`, a parameter `a` of type `[]Job` (method/type names are unrelated), and result type `*M_3`. Arguments are always passed by value. An *interface* specifies a set of methods; a type with a superset of methods *implements* the interface *implicitly*.

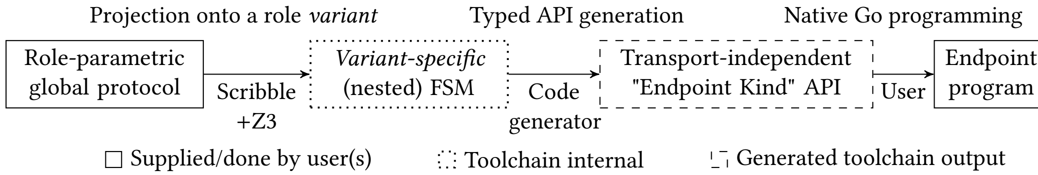


Fig. 3. Main toolchain stages: role-parametric global protocol specification, projection onto a role variant, and distributed Endpoint API generation.

```
type Bar interface { m1(); m2() }                func (b *Baz) m1() { }     func (b *Baz) m2() { }
```

The left side defines an *interface type* `Bar`; the right side implements it for the base type `Baz`.

Package aliases. It is useful to note that Go allows packages (e.g., our generated APIs) to be imported under an alias. This feature allows users of our APIs to locally alias the default generation names, e.g., `import S "github.com/.../pget/Proto1/S_1to1"` aliases `S_1to1` as `S`.

3.2 Distributed, Role-Parametric MPST for Go: Overall Methodology – Pget

We demonstrate our framework by using our toolchain, depicted in Fig. 3, to work through Pget (§ 1.1). For practical protocol specifications, we implement our new theory of role-parametric MPST as an extension to Scribble (http://www.scribble.org/), an existing protocol language based on standard MPST [Coppo et al. 2016]. From the spec, our toolchain generates lightweight APIs that safely prescribe the I/O behaviour of each *role variant* (endpoint *kind*) as a *whole*, i.e., by capturing the causality between I/O actions conducted over otherwise separate underlying channels.

Global protocol. The basic scenario comprises a Master (`M`) coordinating `K` Fetchers (`F`) to download a file from an HTTP Server (`S`). The original project[2] upon which Pget is based implements the former two, to interoperate with standard third party Web servers (e.g., Apache). A *global* protocol, however, specifies the overall application from a neutral perspective: provided the interaction structure can be expressed in terms of (MPST-based) message passing, the details of how any individual endpoint may be implemented remain abstract at this level. This allows for the specification of multiparty applications formed (or partly formed) by a composition of smaller services (e.g., traditional RPCs), similarly to the role of the HTTP server here.

Fig. 4 (top) lists a global protocol `Pget` written in our extended Scribble. We flesh out the description from § 1.1 but keep certain aspects simple for conciseness; subsequent examples will demonstrate further features. We capture the channel mobility in `Pget` using *session-typed* channel passing, called session *delegation* in the literature. The parameterised communication structure in this example is also representative of protocols in other applications (e.g., § 6.2).

The protocol declares the three base *role names* `M`, `F` and `S`. An asynchronous *interaction* is written, e.g., `Job from M to F[1,K];`, where `M` is the *sender*-side, and `F[1,K]` the *receiver*-side; `F[1,K]` stands for the set of `F` in the inclusive, non-empty interval $[1, \texttt{K}]$, where the value of `K` is to be supplied when the session is initiated at *run-time*. By default, `K` is taken to be in $\mathbb{N}_{\geq 1}$: our well-formedness conditions (§ 4.5) determine that the only *valid* instantiations of `K` are values ≥1 (specifically, well-formedness dictates that every interval must be non-empty); the validity of concrete parameter values is checked at run-time. `Job` is the *message signature*, declared in the Scribble module by, e.g.,

```
sig <go> "messages.Job" from "github.com/.../pget/messages" as Job;
```

where `messages.Job` is a Go data type that implements the Scribble API for data serialisation. We omit the similar declarations for the other messages. All together, this interaction specifies a

```
global protocol Pget(role M, role F, role S) {
  Head from F[1]   to S;  Res    from S      to F[1];   // (1) Obtain metadata from Server
  Meta from F[1]   to M;  Job    from M      to F[1,K]; // (2) Allocate Fetcher download tasks
  Get  from F[1,K] to S;  Res    from S      to F[1,K]; // (3) Perform downloads
  Data from F[1,K] to M;  Sync@A from F[1,K] to M;      // (4) Gather data and control channels
} // Sync@A is the local type projection of Sync onto A, i.e., a delegation
global protocol Sync(role A, role B) { choice at A { Done from A to B; } // Choice: terminate B (i.e., Fi) or ...
                                                    or { ... } }
```

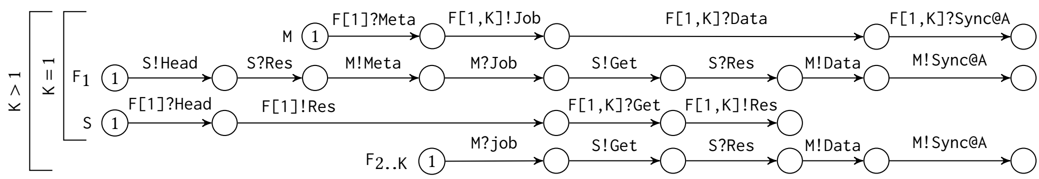


Fig. 4. Pget example from § 1.1 in our extended Scribble: (top) role-parametric global protocol; (bottom) the projections onto each role variant, M, F_1 and $F_{2..k}$, represented as communicating FSMs.

scatter of Job messages (possibly with different values) from the single sender to the K receivers. Similarly, `Get from F[1,K] to S;` specifies a *gather* of K Get messages from the Fs by the single S. Singleton-indexed scatters/gathers coincide as a basic point-to-point interaction.

The message signature of the *delegation* action is Sync@A (adopting the syntax of Scalas et al. [2017]), which denotes passing a *channel* for the A endpoint in the Sync protocol (obtained through projection; see below). For clarity, we name M as A and F as B in Sync (M and F could be reused); and give only the case for terminating the B/F goroutine by sending a Done on the delegated channel.

Projection. The distinct behaviours associated with each role name, i.e., the *role variants*, are inferred from how the role names are *indexed* and used in the protocol body. A role name that is never indexed is implicitly indexed over a singleton constant interval (whose value is irrelevant), as is the case for M and S. Our toolchain infers from the indices that the definition of Pget induces *four* role variants, i.e., four kinds of endpoints: M, F_1, $F_{2..K}$ and S. Fig. 4 (bottom) depicts the projection of Pget onto each: our implementation uses a representation of our index-parameterised local types (§ 4.2) based on *communicating finite state machines* [Brand and Zafiropulo 1983; Deniélou and Yoshida 2012] that correspond straightforwardly to the syntactic types. In our setting, the FSMs communicate via *scatter/gather* I/O (subsuming basic point-to-point messaging), and may feature *nesting* of FSMs inside states (demonstrated in § 3.3). The toolchain also determines these variants form two valid *families*: one has M, F_1 and S ($K = 1$), and the other has all ($K > 1$).

The initial states are marked 1. For instance, in the FSM for M, the first action F[1]?Meta receives the Meta message from F[1], followed by F[1,K]!Job that scatters Jobs to the K Fs. Then M waits until it has gathered a Data from each F, and likewise the delegated control channel of type Sync@A.

API types generation. The purpose of the API generation is to capture a projection as Go type definitions to guide programming of the target variant, and impart safety assurances through a combination of type checking and the functionality of the underlying generated code. It is possible to generate various kinds of API, suited to different programming styles—a benefit of our distributed framework (cf. previous "monolithic" approaches Ng et al. [2015]; Yoshida et al. [2010]) is that different endpoints could be separately implemented using different APIs: we present the most direct API generation from a projection, that is close to channel-based programming in common practices (e.g., TCP sockets, Go channels) and to the session π-calculi in MPST formalisms.

State type (with nested peer/action types)			**Method name and signature** (parameters, result type)	
State	Peer(s)	I/O action	Message label/values, aux. functions	Successor
`M_1`	`F_1`	`Receive`	`Meta(a *Meta)`	`*M_2`
`M_2`	`F_1toK`	`Scatter`	`Job(a []Job)`	`*M_3`
`M_3`	`F_1toK`	`Gather`	`Data(a []Data)`	`*M_4`
`M_4`	`F_1toK`	`GatherAndSpawn`	`Sync_A(run func(*A_1) End_A)`	`End_M`

```
func mainM(req HttpReq, K int) {
  proto := Pget.New()
  M := proto.M.Kgt1.New(K) // API for K>1
  ss1 := shm.Listen(8888+1); defer ss1.close()
  go mainF1(req, 8888+1)
  M.F_1.Accept(ss1)
  for i := 2; i <= K; i++ {
    ssi := shm.Listen(8888+i); defer ssi.close()
    go mainF_2toK(req, 8888+i)
    M.F_2toK.Accept(i, ssi) // Supported by K>1 API
  M.run(runM) // runM: func(*M_1) End_M
} }
```

```
func runM(m *M_1) End_M {
  var meta Meta; var data Data
  // F[1]?Meta. F[1,K]!Job. F[1,K]?Data. F[1,K]?Sync@A
  return m.F_1    .Receive        .Meta(&meta).
           F_1toK.Scatter        .Job(split(&meta)).
           F_1toK.Reduce         .Data(&data, agg).
           F_1toK.GatherAndSpawn.Sync_A(runA)
}

func runA(a *A_1) End_A {
  return a.B.Send.Done() // Just do Done, for brevity
}
```

Fig. 5. (top) Go API types and I/O method signatures generated for `M` in Pget; (bottom) an `M` endpoint implementation using the generated API.

In short, the API generation takes the FSM for a target role *variant* and (i) reifies each state as a *state-specific* Go type, that (ii) offers a generated I/O method for each of the *transitions* from that state; the result type of each I/O method is set to the *successor* state of that transition. We refer to instances of the state-specific types as *state channels*, and they are created only by the API itself. A state channel API is basically an interlinked set of lightweight, variant- and state-specific type wrappers that abstract from the concrete I/O actions on the underlying channels (Go channels, TCP, etc.) to the various participants of a multiparty communication session.

Fig. 5 (top) summarises the state channel API generated for `M`. On the left, 'State' is the "top-level" type for each protocol (FSM) state. 'Peer' is a type that denotes the valid interaction peers at each state, accessed as a field of State; similarly the valid 'I/O action's are also denoted by types accessed as fields of Peer. On the right, the valid message types for each action are offered as methods on the action types, taking the message values as parameters, and resulting in the successor state type. The various actions (e.g., `Receive`, `Scatter`) and parameters are generated based on the FSM state.

As an example, assuming variables `m` and `meta` of the initial state type `M_1` and message type `meta`, respectively, the first I/O action in an `M` program may be guided by the API as:

`m.F_1.Receive.Meta(&meta)` which can be read as: on channel `m`, do $\mathtt{F}_1$`?Meta`.

Since the result type of I/O methods is used for successor states, input methods like `Receive`/`Gather` are generated to store the deserialised message values into the pointer arguments (e.g., `meta`), following idiomatic usage of standard Go APIs (e.g., `encoding/gob`). The alternative of returning a pair of the successor state and the deserialised values hinders fluent call-chaining. Variable declarations in Go allocate memory initialised to zero values (and are thus safe to read).

We highlight that the I/O method parameters relate only to messages: all index computations and mappings to underlying channels are internalised within the API from the source specification. For simplicity, we use the default type/method naming as illustrated; users may instead use Go package/type aliases (each state has a separate subpackage; cf. § 3.1) in the local program, or supply name annotations in the protocol—i.e., specific naming schemes are not a crucial detail.

Endpoint programming. Fig. 5 gives an example Go implementation of `M` using the API generated as above. We assume Go type definitions (e.g., `Get`, `Res`) for each message signature as described earlier, and a `HttpReq` helper type that holds the various field values of a HTTP request.

ENDPOINT INITIATION An endpoint implementation typically starts by establishing a new session for the target protocol, signified by instantiating the generated API frontend type: here, `proto` of type `Pget`. This is used to create a new *Endpoint* by using the appropriate constructor from a generated "menu" of nested type functions: e.g., line 3 in Fig. 5 uses the constructor for `M` under the `K_gt1` (`K` > 1) family. An incompatible `K` argument for this family is a run-time error: a check on the implicit constraint (derived from the protocol) is built into the generated method (§ 5.3). An Endpoint is first used to establish communication links to its peers by the generated connection methods `Accept` (lines 6 and 10) and `Dial` (illustrated below), similarly to standard Socket APIs (e.g., `tcp` or `unix` via the `net` package), with the additional option to use shared memory Go channels (`shm` package) as a transport; in Pget, for instance, the Master and the Fetchers communicate via shared memory, as indicated by the usage of the `shm` package on Lines 4 and 8. The `K` > 1 API selected for `M` in this code supports (i.e., allows by static typing) the `Accept` (and `Dial`) method for $\texttt{F}_{\texttt{2..K}}$ (line 10); whereas the `K` = 1 API has connection methods only for $\texttt{F}_1$.
After initiating the session, we use a generated `run` method on the Endpoint to conduct the protocol by supplying a `func(*M_1) End_M`, where `M_1` is the *initial* state channel type of this endpoint, and `End_M` is the *terminal* type. We note the result is set to the `End` type even for non-terminating endpoints (i.e., persistent *sessions*)—since no generated I/O method will actually return a state channel of this type, this signifies the function should be non-terminating.

PROTOCOL IMPLEMENTATION Intuitively from an FSM view, an implementation of the `run` argument function using the state channel API must observe one simple usage condition: *on the current state channel, call* exactly one *I/O method to obtain the next, up to the terminal state* (if any). Following this, the implementation, e.g., `runM` (line 14), is thus guided by the static type of each state channel as the programmer works through the protocol. For a given session instance, the only way to obtain a value from the API that statically satisfies the `End` result type of a (terminating) endpoint is to reach and perform a generated I/O method that corresponds to a terminal transition.
We have used the API in a concise call-chaining style; the user may also use the generated types in more explicitly imperative (e.g., protocol steps as sequenced statements) or "functional" (e.g., via functions with state type parameters and result) styles, interleaved with other application operations as needed. The `Reduce` method on line 19 is an additionally generated convenience variant of the basic `Gather` (Fig. 5, top). We omit the simple definitions of functions `split` and `agg`.

TRANSPORT ABSTRACTION AND DELEGATION Endpoint programs for each variant are implemented in a similar fashion. Assuming an `F1` Endpoint object created using the generated API, we may find in a preamble for $\texttt{F}_1$:

```
F1.M.Dial(shm.Client, "localhost", portM); F1.S.Dial(tcp.Client, req.Host, req.Port)
```

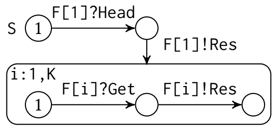
`F1` is used to connect (`Dial`) to `M` and `S` on shared memory and TCP transports, respectively. Starting from the initial state channel (below, `f`), the programmer can rely on the API to guide the way through the multiparty protocol for $\texttt{F}_1$ (cf. its FSM, Fig. 4) as a whole, correctly dispatching the interleaved I/O operations with `M` and `S` on the underlying `shm` and `tcp` channels:

```
// Assuming vars req:HttpReq, res:Res, job:Job, etc., F₁ does: S!Head. S?Res. M!Meta. M?Job. ...
f. S.Send.Head(req.Head()). S.Receive.Res(&res). M.Send.Meta(res.Meta()). M.Receive.Job(&job). ...
```

Our API generation takes advantage of cheap goroutine spawning to offer various convenience methods for delegations. In the `run` method of the delegation sender, i.e., $\texttt{F}_1$:

```
  // New Sync session   // Spawns B goroutine       // M!Sync@A.end -- i.e., delegate 'a' to M
... proto := Sync.New(); a := proto.Shm.A.New(runB);   return f8.M.Send.Sync_A(a)
```

The second step is a Scribble-Go API facility for establishing shared memory sessions: the `New` constructs an `A` endpoint of a new session for the `Sync` protocol (Fig. 4), while spawning a goroutine




```
1 func runS(s *S_1) End {
2   var head *Head
3   return s.F_1.Receive.Head(head).
4             F_1.Send.Res(Res.New(head)).
5             Foreach(nested) }
```

```
7 func nested(i int, s *S3_1) S3_End {
8   var get *Get
9   return s.F_i.Receive.Get(get).
10            F_i.Send.Res(Res.New(get))
11 }
```

Fig. 6. Projection and example user code for S in the revision of Pget using foreach.

for the implementation function supplied for each of the other endpoints (i.e., runB for B); shm channels are implicitly created between each endpoint. Assuming f8 is of the penultimate state type for F_1, the Send then *delegates* the state channel a to M, satisfying the local type M!Sync@A. The GatherAndSpawn in M (Fig. 5, line 20) is generated for receiving *channels*: it implicitly spawns the supplied function, typed from the received state to End, as a goroutine for each received channel.

<u>State channel linearity and safety guarantees</u> The use-exactly-once (i.e., *linear* use) condition of state channel APIs means a program should never *re*use a state channel instance: as a default, the API generation inlines minimal run-time checks against repeat channel use into the API, though our examples illustrate how call-chaining may help avoid linearity errors by keeping intermediate channel values implicit. But *regardless* of channel linearity, a generated API guarantees that an endpoint implementation *never* performs a non-compliant I/O action w.r.t. to the run-time instantiation of the parameterised protocol, up to premature termination (e.g., failures). We discuss linearity, options for *static* linearity, and our safety guarantees in § 5.4.

3.3 Pget – Revised using foreach (Role-Parametric Subprotocols as Nested FSMs)

Like the original program, an MPST-based (re-)implementation of the client side of Pget (M, F_1 and $F_{2..K}$) is interoperable with a third-party S such as Apache. However, our framework equally allows to implement an S that would be interoperable with the original client (and our Scribble client).

The specification in Fig. 4 has: `Get from F[1,K] to S; Res from S to F[1,K];`. As depicted there, the projection onto S results in a gather from all Fs (F[1,K]?Get) and a scatter to all Fs (F[1,K]!Res). In practice, the more desirable behaviour is for S to serve the Get-Res exchange with each F *concurrently*. This may be specified via our foreach extension to Scribble, that allows to express a form of *role-parametric subprotocols*: we can replace line 4 in Fig. 4 by

```
foreach F[i:1,K] { Get from F[i] to S; Res from S to F[i]; }
```

Fig. 6 depicts the projection by our toolchain onto S: the *default* behaviour is to repeat the nested FSM for i:1..K in sequence. The same FSMs and APIs are generated for F_1 and $F_{2..K}$ as in Fig. 4.

Fig. 6 (right) gives an implementation of S using the default foreach API generation. The basic API generation for a state *s* with a nested FSM is to generate a Foreach method, that on entering *s* first executes the subprotocols to completion: it takes the nested behaviour as a first-class function, and performs it sequentially over the parameter range [1,K] (implicit within the generated API). In general, Foreach then returns an intermediary value for performing the transition out of *s*; in this example, it directly returns End. When parameterised variants within a foreach do *not* interact with each other, however, an additional method is generated that alternatively performs the subprotocols in *parallel*. As desired of S above, this allows by replacing lines 3–5 in Fig. 6:

```
return s. F_1.Receive.Head(head). F_1.Send.Res(Res.New(head)). Parallel(nested)
```

The Parallel method spawns a separate nested goroutine for each parameter value.

Further examples. We demonstrate protocol *branching* and *recursion* in a range of later examples, in formal notation (e.g., Ex. 4.4, Ex. 4.8 in § 4.2) and our Go APIs (e.g., Fig. 13 in § 5.3). An implementation of F_1 and other larger examples are in the Supplement[4] (e.g., § I.1.3, § I.2, § IV.1.2).

$$E \in \mathbb{E} ::= E_1 \texttt{+} E_2 \mid E_1 \texttt{-} E_2 \mid a \mid k \qquad D \in \mathbb{D} ::= E_1 \,..\, E_2 \qquad a \in \mathbb{A} \qquad k \in \mathbb{K} \qquad i \in \mathbb{I} \qquad x \in \mathbb{E} \cup \mathbb{I}$$
$$G \in \mathbb{G} ::= r_1[x_1] \to r_2[x_2] : \{\ell_j \,.\, G_j\}_{j \in J} \mid \texttt{foreach}\, R\{i_j : D_j\}_{j \in J}\, \texttt{do}\, G_1 \,;\, G_2 \mid \texttt{cont} \mid \texttt{rec}\, X\, G \mid X \mid \texttt{end}$$
$$L \in \mathbb{L} ::= r[x]\,!\{\ell_j \,.\, L_j\}_{j \in J} \mid r[x]\,?\{\ell_j \,.\, L_j\}_{j \in J} \mid \texttt{foreach}\, R\{i_j : D_j\}_{j \in J}\, \texttt{do}\, L_1 \,;\, L_2 \mid \texttt{cont} \mid \texttt{rec}\, X\, L \mid X \mid \texttt{end}$$

Fig. 7. Syntax of rank expressions ($E \in \mathbb{E}$), intervals ($D \in \mathbb{D}$), global types ($G \in \mathbb{G}$), and local types ($L \in \mathbb{L}$)

4 THEORY

Our new theory generalises the original MPST [Coppo et al. 2015; Honda et al. 2016]. It consists of the following contributions: § 4.1 – an abstract algebra of *ranks* to index role names, which subsumes index domains in existing parameterised MPST approaches; § 4.2 – languages of parameterised global types and local types, to specify communication patterns among indexed roles from a global perspective and a local perspective, using a new `foreach` construct; § 4.3 – the first static *inference procedure* for role variants; § 4.4 – a new projection operator that produces local types for role variants, based on a global type; and § 4.5 – theorems that certify role variant inference is decidable, checking well-formedness is decidable, and projection is correct (i.e., the set of local types projected from a well-formed global type is equivalent to the global type; this implies safety).

4.1 Roles and Ranks

Roles. Let $\mathbb{R}$ denote the set of all *role names*, ranged over by r (and R over sets of role names). Every role name identifies a *role* that *individuals* (i.e., endpoint programs, e.g., goroutines) enact in a protocol. For instance, the role names in the Pget protocol are `M` for Master, `F` for Fetchers, and `S` for Server. Our theory allows every *single role* to be enacted by *multiple individuals*.

Ranks. Let $\mathbb{A}$ denote the set of all *ranks*, ranged over by a. Every rank identifies an individual among the possibly many that enact the same role (cf. ranks in MPI; principals in Wysteria [Rastogi et al. 2014]), through *indexed role names*. For instance, `F[3]` identifies the third Fetcher.

Our theory is parametric in $\mathbb{A}$, meaning we do not fix a specific set of ranks. Instead, more abstractly, the only structure we assume of $\mathbb{A}$ is the existence of an operator $+$, a constant 0, and relations $\leq$ and $<$, such that: $\langle \mathbb{A}, +, 0 \rangle$ is a torsion-free abelian group; $\langle \mathbb{A}, \leq \rangle$ is a partially ordered set; $\langle \mathbb{A}, < \rangle$ is a strictly totally ordered set; $+$ preserves $\leq$ and $<$; first-order formulas over $\langle \mathbb{A}, +, 0, \leq \rangle$ are decidable; and the set of ranks between any ranks a_1 and a_2 under $\leq$ (i.e., $\{a \mid a_1 \leq a \leq a_2\}$) is finite and enumerable. If these conditions are satisfied, we call $\langle \mathbb{A}, +, 0, \leq, < \rangle$ a *rank structure*. The Supplement,[4] § II.1 motivates the need for these conditions.

Example 4.1 (1d). The set of all integers $\mathbb{Z}$, with the standard integer addition for $+$, and with the standard non-strict and strict integer orders for $\leq$ and $<$, is a rank structure; $\langle \mathbb{A}, +, 0, \leq \rangle$ yields linear integer arithmetic, which is decidable.

Example 4.2 (2d). The set of all pairs of integers $\mathbb{Z} \times \mathbb{Z}$, with *coordinate-wise addition* for $+$, with the non-strict *product order* for $\leq$, and with the strict *lexicographic order* for $<$, is a rank structure; $\langle \mathbb{A}, +, 0, \leq \rangle$ can be encoded in linear integer arithmetic, which is decidable. $\mathbb{A} = \mathbb{Z} \times \mathbb{Z}$ enables indexing role names with *2d coordinates*, for matrix and mesh protocols; see Ex. 4.6, 4.7.

4.2 Global Types and Local Types

Preliminaries. Global types specify communication patterns among a possibly unknown number of individuals from a global perspective. We start with some preliminaries.

- We assume a set $\mathbb{K} = \{\mathsf{k}_1, \mathsf{k}_2, \ldots\}$ of all *parameters*, ranged over by k.
- We define the set $\mathbb{E}$ of all *rank expressions*, ranged over by E (Fig. 7, first line). If a rank expression contains parameters, it is *open*; otherwise, it is *closed*.

- We define the set $\mathbb{D}$ of all *intervals*, ranged over by D (Fig. 7, first line).
- We assume a set $\mathbb{I} = \{\mathtt{i}_1, \mathtt{i}_2, \ldots\}$ of all *index variables*, ranged over by i. We use index variables to iterate over intervals, denoted as $i\,{:}\,D$. Let $\mathbb{E} \cup \mathbb{I}$ denote the set of all *indices*, ranged over by x.

Global types. Fig. 7, second line, shows the syntax of global types. $r_1[x_1] \to r_2[x_2] : \{\ell_j \,.\, G_j\}_{j \in J}$ denotes an asynchronous *communication* of a message labelled as ℓ_j from *sender* $r_1[x_1]$ to *receiver* $r_2[x_2]$, for $j \in J$ (chosen by the sender), followed by G_j; as the syntax of message labels is irrelevant in our theory, we leave it unspecified. We omit curly brackets if J is a singleton; also, if a role is enacted by only one individual, we omit its index (e.g., we write `M` instead of `M[0]` for Master). `rec` X G denotes *recursion*; `end` denotes termination.

iter.	$\mathtt{i}_1$	$\mathtt{i}_2$	body after substitution
1	1	2	$\mathtt{W[1]} \to \mathtt{W[2]} : \mathtt{Val}\ .\ \mathtt{cont}$
2	2	3	$\mathtt{W[2]} \to \mathtt{W[3]} : \mathtt{Val}\ .\ \mathtt{cont}$
…	…	…	…
7	8-1	8	$\mathtt{W[8-1]} \to \mathtt{W[8]} : \mathtt{Val}\ .\ \mathtt{cont}$

$$G_{\text{Pipe}} = \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1{:}1..\mathtt{k}\text{-}1, \mathtt{i}_2{:}2..\mathtt{k}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]} : \mathtt{Val}\ .\ \mathtt{cont})\ ;\ \mathtt{end}$$

Fig. 8. Table ($\mathtt{k} = 8$) for the iteration domain in the Pipeline global type

`foreach` $R\{i_i\,{:}\,D_j\}_{j\in J}$ `do` G_1 ; G_2, the key novelty of our language, denotes a *loop* of the communications specified in *body* G_1, followed by *continuation* G_2; `cont` indicates the loop should continue with the next iteration. The *iteration domain* of `foreach` is specified by $R\{i_j\,{:}\,D_j\}_{j\in J}$, where R denotes a non-empty set of role names, and where every D_j has the same length; it essentially constitutes a "table", where "columns" correspond to index variables, "rows" to iterations, and the "cell" in column i_j, row u, contains the u-th rank in D_j (sorted by <). The intervals are iterated over in lock-step: the idea is that in the u-th iteration of the loop, at run-time, individuals communicate with each other as specified in G_1 after substituting $r[a]$ for $r[i_j]$, for every $r \in R$, and where a is the corresponding rank in the table. For instance, Fig. 8 shows the table for the iteration domain in the Pipeline global type. By definition (i.e., the conditions on rank structures, plus every interval has a lower and upper bound), every interval is finitely enumerable.

The bounded "counting" aspect of our `foreach` is inspired by dependent type theories and the primitive recursion operator used in previous work (§ 2.2). However, a unique feature of our *MPST-oriented* `foreach` is that it essentially iterates over indexed role names (`W[1]`, `W[2]`, …) instead of over "naked" indices (`1`, `2`, …; cf. primitive recursion). Leveraging this role-based information is key to facilitating the static, decidable inference of role variants (§ 4.3), projection (§ 4.4), and checking condition 3 of well-formedness (§ 4.5).

Remark 1. An iteration domain $\{r_1,\ldots,r_n\}\{i_1\,{:}\,D_1,\ldots,i_m\,{:}\,D_m\}$ can equivalently, and closer to our extended Scribble notation, be represented as a sequence $r_1[i_1\,{:}\,D_1], r_1[i_2\,{:}\,D_2], \ldots, r_n[i_m\,{:}\,D_m]$, where n and m are unrelated. Our present notation is more convenient to deal with in proofs.

Example 4.3 (Pget). Let `k` represent the number of Fetchers in the Pget protocol (§ 3.2). The following global type specifies the first half of the Pget protocol ($\mathbb{A} = \mathbb{Z}$): $G_{\text{Pget}} =$

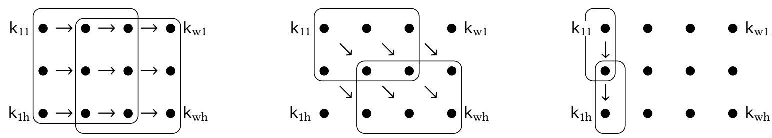
$\mathtt{F[1]} \to \mathtt{S} : \mathtt{Head}\ .\ \mathtt{S} \to \mathtt{F[1]} : \mathtt{Res}\ .\ \mathtt{F[1]} \to \mathtt{M} : \mathtt{Size}\ .\ \mathtt{foreach}\ \mathtt{F\{i{:}1..k\}}\ \mathtt{do}\ (\mathtt{M} \to \mathtt{F[i]} : \mathtt{Range}\ .\ \mathtt{cont})\ ;\ \ldots$

Example 4.4 (Ring). Let `k` represent the number of Workers in the Ring protocol (§ 2.2). The following global type specifies the Ring protocol, extended with branching and recursion ($\mathbb{A} = \mathbb{Z}$):

$$G_{\text{Ring}} = \mathtt{rec}\ \mathtt{X}\ \mathtt{W[1]} \to \mathtt{W[2]} : \left\{ \begin{array}{l} \mathtt{Nx}\ .\ \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1{:}2..\mathtt{k}\text{-}1, \mathtt{i}_2{:}3..\mathtt{k}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]} : \mathtt{Nx}\ .\ \mathtt{cont})\ ;\ (\mathtt{W[k]} \to \mathtt{W[1]} : \mathtt{Nx}\ .\ \mathtt{X}) \\ \mathtt{Dn}\ .\ \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1{:}2..\mathtt{k}\text{-}1, \mathtt{i}_2{:}3..\mathtt{k}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]} : \mathtt{Dn}\ .\ \mathtt{cont})\ ;\ (\mathtt{W[k]} \to \mathtt{W[1]} : \mathtt{Dn}\ .\ \mathtt{end}) \end{array} \right\}$$

Example 4.5 (Fibonacci). The following global type specifies a Fibonacci-`k` protocol ($\mathbb{A} = \mathbb{Z}$):

$$G_{\text{Fib}} = \mathtt{foreach}\ \mathtt{Fib}\{\mathtt{i}_{(-2)}{:}1..\mathtt{k}\text{-}2, \mathtt{i}_{(-1)}{:}2..\mathtt{k}\text{-}1, \mathtt{i}{:}3..\mathtt{k}\}\ \mathtt{do}\ (\mathtt{Fib}[\mathtt{i}_{(-2)}] \to \mathtt{Fib[i]} : \mathtt{Val}\ .\ \mathtt{Fib}[\mathtt{i}_{(-1)}] \to \mathtt{Fib[i]} : \mathtt{Val}\ .\ \mathtt{cont})\ ;\ \mathtt{end}$$



(a) Horizontal wave (b) Diagonal wave (c) Column pipeline

Fig. 9. Basic mesh communication patterns (Ex. 4.7)

Example 4.6 (Hadamard). Let $\mathtt{k}_{11}$ and $\mathtt{k}_{\mathrm{wh}}$ represent the top-left and the bottom-right of 2d matrices A, B, and C. The following global types specifies a protocol to compute the Hadamard product (i.e., coordinate-wise product) of A and B as C ($\mathbb{A} = \mathbb{Z} \times \mathbb{Z}$):

$$G_{\mathrm{Had}} = \mathtt{foreach}\ \{\mathtt{A,B,C}\}\{\mathtt{i}:\mathtt{k}_{11}..\mathtt{k}_{\mathrm{wh}}\}\ \mathtt{do}\ (\mathtt{A[i]} \to \mathtt{C[i]} : \mathtt{Val}\ .\ \mathtt{B[i]} \to \mathtt{C[i]} : \mathtt{Val}\ .\ \mathtt{cont})\ ;\ \mathtt{end}$$

Example 4.7 (Mesh). Let $\mathtt{k}_{11}$, $\mathtt{k}_{1\mathrm{h}}$, $\mathtt{k}_{\mathrm{w}1}$, and $\mathtt{k}_{\mathrm{wh}}$ represent the top-left, the bottom-left, the top-right, and the bottom-right of a 2d mesh. The following global types (message labels omitted), three of which are visualised in Fig. 9 for a 4×3 mesh, specify five basic mesh communication patterns: horizontal wave, diagonal wave, column pipeline, 2d scatter, 2d gather.

$$\begin{aligned}
G_{\mathrm{HWave}} &= \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1:\mathtt{k}_{11}..\mathtt{k}_{\mathrm{wh}}\text{-}(1,0),\mathtt{i}_2:\mathtt{k}_{11}\text{+}(1,0)..\mathtt{k}_{\mathrm{wh}}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]}\ .\ \mathtt{cont})\ ;\ \mathtt{end}\\
G_{\mathrm{DWave}} &= \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1:\mathtt{k}_{11}..\mathtt{k}_{\mathrm{wh}}\text{-}(1,1),\mathtt{i}_2:\mathtt{k}_{11}\text{+}(1,1)..\mathtt{k}_{\mathrm{wh}}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]}\ .\ \mathtt{cont})\ ;\ \mathtt{end}\\
G_{\mathrm{ColPipe}} &= \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1:\mathtt{k}_{11}..\mathtt{k}_{1\mathrm{h}}\text{-}(0,1),\mathtt{i}_2:\mathtt{k}_{11}\text{+}(0,1)..\mathtt{k}_{1\mathrm{h}}\}\ \mathtt{do}\\
&\qquad (\mathtt{W[i_1]} \to \mathtt{W[i_2]}\ .\ \mathtt{cont})\ ;\ \mathtt{W[k_{1h}]} \to \mathtt{W[k_{11}]}\ .\ \mathtt{end}\\
G_{\mathrm{2dSca}} &= \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}:\mathtt{k}_{11}..\mathtt{k}_{\mathrm{wh}}\}\ \mathtt{do}\ (\mathtt{M} \to \mathtt{W[i]}\ .\ \mathtt{cont})\ ;\ \mathtt{end}\\
G_{\mathrm{2dGat}} &= \mathtt{foreach}\ \mathtt{W}\{\mathtt{i}:\mathtt{k}_{11}..\mathtt{k}_{\mathrm{wh}}\}\ \mathtt{do}\ (\mathtt{W[i]} \to \mathtt{M}\ .\ \mathtt{cont})\ ;\ \mathtt{end}
\end{aligned}$$

Remark 2. Although our `foreach` operator for global types unrolls iterations of its body sequentially in terms of its index values, it maintains the concurrency characteristics of MPST. E.g., in standard MPST, the two interactions in $\mathtt{A} \to \mathtt{B} : \mathtt{Foo}\ .\ \mathtt{C} \to \mathtt{D} : \mathtt{Bar}$ are concurrent since the roles in each are independent; this remains the case if such a fragment occurs inside a `foreach`, e.g., the `A`/`B` action of the final iteration could potentially occur before the `C`/`D` of the first iteration.

In addition to such "latent" concurrency, a global `foreach` may be elided from the *local* type by projection depending on the communication pattern. For instance, none of the Worker local types in the Pipeline protocol (shown in the next paragraph) has `foreach`, contrasting the global type in Fig. 8. This observation is more pronounced when extended to a Recursive Pipeline protocol, $\mathtt{rec}\ X\ (\mathtt{foreach}\ \mathtt{W}\{\mathtt{i}_1:1..\mathtt{k}\text{-}1,\mathtt{i}_2:2..\mathtt{k}\}\ \mathtt{do}\ (\mathtt{W[i_1]} \to \mathtt{W[i_2]} : \mathtt{Val}\ .\ \mathtt{cont})\ ;\ X)$, which allows multiple Worker pairs (participating in different recursive calls) to communicate concurrently.

Our implementation also supports runtime parallelisation of `foreach` as an optimisation, when parameterised variants do not interact (demonstrated in § 3.3).

Local types. Fig. 7, third line, shows the syntax of local types. $r[x]\ !\{\ell_j\ .\ L_j\}_{j\in J}$ denotes the *send* of a message labelled as ℓ_j to receiver $r[x]$, for $j \in J$ (chosen by the sender), followed by the actions specified in L_j. Symmetrically, $r[x]\ ?\{\ell_j\ .\ L_j\}_{j\in J}$ denotes the *receive* of a message labelled as ℓ_j from sender $r[x]$. For instance, the local types for $\mathtt{k} = 3$ Workers in the Pipeline protocol are:

$$L^{\mathtt{W[1]}}_{\mathrm{Pipe}} = \mathtt{W[2]}\ !\ \mathtt{T}\ .\ \mathtt{end} \qquad L^{\mathtt{W[2]}}_{\mathrm{Pipe}} = \mathtt{W[1]}\ ?\ \mathtt{T}\ .\ \mathtt{W[3]}\ !\ \mathtt{T}\ .\ \mathtt{end} \qquad L^{\mathtt{W[3]}}_{\mathrm{Pipe}} = \mathtt{W[2]}\ ?\ \mathtt{T}\ .\ \mathtt{W[4]}\ !\ \mathtt{T}\ .\ \mathtt{end}$$

The Supplement,[4] § II.2 contains more example local types, for the same protocols as above.

Syntactic sugar. Fig. 10 shows syntactic sugar for `foreach` in global types and local types. $\xrightarrow{*}$ expands to an *all-to-all* global type; it demonstrates `foreach` nesting. $\xRightarrow{*}$ expands to a *pairings* global type. Note that while senders may have multiple labels to choose from (if $|J| > 1$), each of

$$
\begin{array}{rcl}
r_1[D_1] \xrightarrow{*} r_2[D_2] : \{\ell_j \,.\, G'\}_{j\in J} & \triangleq & \texttt{foreach}\ r_1\{\texttt{i}_1 : D_1\}\ \texttt{do} \\
 & & \quad (\texttt{foreach}\ r_2\{\texttt{i}_2 : D_2\}\ \texttt{do}\ (r_1[\texttt{i}_1] \to r_2[\texttt{i}_2] : \{\ell_j \,.\, \texttt{cont}\}_{j\in J})\ ;\ \texttt{cont})\ ;\ G' \\
r_1[D_1] \xRightarrow{*} r_2[D_2] : \{\ell_j \,.\, G'\}_{j\in J} & \triangleq & \texttt{foreach}\ \{r_1, r_2\}\{\texttt{i}_1 : D_1, \texttt{i}_2 : D_2\}(r_1[\texttt{i}_1] \to r_2[\texttt{i}_2] : \{\ell_j \,.\, \texttt{cont}\}_{j\in J})\ ;\ G' \\
r_1 \xrightarrow{1} r_2[E_1 .. E_2] : \{\ell_j \,.\, G_j\}_{j\in J} & \triangleq & r_1 \to r_2[E_1] : \{\ell_j \,.\, \texttt{foreach}\ r_2\{\texttt{i} : E_1\texttt{+1} .. E_2\}\ \texttt{do}\ (r_1 \to r_2[\texttt{i}] : \ell_j \,.\, \texttt{cont})\ ;\ G_j\}_{j\in J} \\
r \xRightarrow{1} [E_1 .. E_2] : \{\ell_j \,.\, G_j\}_{j\in J} & \triangleq & r[E_1] \to r[E_1\texttt{+1}] : \left\{\begin{array}{l} \ell_j \,.\, \texttt{foreach}\ r\{\texttt{i}_1 : E_1\texttt{+1} .. E_2\texttt{-1}, \texttt{i}_2 : E_1\texttt{+2} .. E_2\}\ \texttt{do} \\ \qquad (r[\texttt{i}_1] \to r[\texttt{i}_2] : \ell_j \,.\, \texttt{cont})\ ;\ G_j \end{array}\right\}_{j\in J} \\
r[D]\ \dagger^*\{\ell_j \,.\, L'\}_{j\in J} & \triangleq & \texttt{foreach}\ r\{\texttt{i} : D\}\ \texttt{do}\ (r[\texttt{i}]\ \dagger\{\ell_j \,.\, \texttt{cont}\}_{j\in J})\ ;\ L' \quad \text{if } \dagger \in \{!, ?\} \\
r[D]\ !^1\{\ell_j \,.\, L_j\}_{j\in J} & \triangleq & r[E_1]\ !\,\{\ell_j \,.\, \texttt{foreach}\ r\{\texttt{i} : E_1\texttt{+1} .. E_2\}\ \texttt{do}\ (r[\texttt{i}]\ !\ \ell_j \,.\, \texttt{cont})\ ;\ G_j\}_{j\in J} \quad \text{if } D = E_1 .. E_2
\end{array}
$$

Fig. 10. Syntactic sugar for global types ($\xrightarrow{*}$, $\xRightarrow{*}$, $\xrightarrow{1}$, $\xRightarrow{1}$) and local types ($\dagger^*$, $!^1$), under $\mathbb{A} = \mathbb{Z}$

these choices has the same continuation G'. This is to syntactically enforce a fundamental rule of interacting parties in a parameterised setting: if a protocol allows separate parties to make independent (inconsistent) choices without additional synchronisation, the continuation of that protocol cannot depend on any of those choices (because parties are not aware of all choices made).

$\xrightarrow{1}$ expands to a *master-slaves* global type, where the master (r_1) chooses a message label from $\{\ell_j\}_{j\in J}$ and communicates a corresponding message to all its slaves (r_2); the distinguished communication from the master to the first slave ensures the master commits to its initial choice. $\xRightarrow{1}$ expands to a *pipeline* global type, where the front Worker chooses a message label from $\{\ell_j\}_{j\in J}$, then corresponding messages are propagated onward. In these two sugars, only one choice is made (in contrast to $\xrightarrow{*}$ and $\xRightarrow{*}$), known to all parties, allowing choice-dependent continuations.

$\dagger^*$ expands to a *send-to-all* ($\dagger = !$) or *receive-from-all* ($\dagger = ?$) local type that corresponds precisely to the projections of (the expansion of) $\xrightarrow{*}$. Similarly, $!^1$ expands to a send-to-all local type that corresponds precisely to the projections of $\xrightarrow{1}$. The difference between $!^*$ and $!^1$ pertains to *the number of choices* made: with $!^*$, the sender *may* choose a *different* message label for every receiver, while with $!^1$, the sender *must* choose the *same* message label for every receiver. (No special local type sugar is needed for the remaining global type sugar, as its projections have no `foreach`.)

Example 4.8 (Syntactic sugar). The global types in Ex. 4.3, 4.4 can be rewritten:

$$G_{\text{Pget}} = \texttt{F[1]} \to \texttt{S} : \texttt{Head} \,.\, \texttt{S} \to \texttt{F[1]} : \texttt{Res} \,.\, \texttt{F[1]} \to \texttt{M} : \texttt{Size} \,.\, \texttt{M} \xrightarrow{*} \texttt{F[1..k]} : \texttt{Range} \,.\, \ldots$$

$$G_{\text{Ring}} = \texttt{rec X}\ (\texttt{W} \xRightarrow{1} \texttt{[1..k]} : \{\texttt{Next} \,.\, \texttt{W[k]} \to \texttt{W[1]} : \texttt{Next} \,.\, \texttt{X},\ \texttt{Done} \,.\, \texttt{W[k]} \to \texttt{W[1]} : \texttt{Done} \,.\, \texttt{end}\})$$

4.3 Role Variants

Role variants. In our theory, *different* individuals that enact a role with *the same* name may have *different* communication behaviours; theoretically, role names are uninterpreted constants, void of semantics. For instance, the front Worker (who only sends), the middle Workers (who both receive and send), and the back Worker (who only receives) in the Pipeline protocol (Fig. 8) have different communication behaviours, but they all enact the same role `W`.

This phenomenon presents a theoretical challenge: neither can we associate a single local type L with a role r (i.e., L can impossibly cover all behavioural variations exhibited by individuals that enact r), nor can we associate a local type with every individual (i.e., the number of individuals can be unknown until run-time). To solve this problem, we introduce the concept of *role variants*: a group of (ranks of) individuals that *both* enact the same role r *and* have "the same" behaviour, in the sense that the behaviour of each of these individuals can be specified by the same local type. For instance, the single local type that specifies the behaviour of every middle Worker is:

$$L_{\text{Pipe}}^{\texttt{W[2..k-1]}} = \texttt{W[self-1]}\ ?\ \texttt{Val} \,.\, \texttt{W[self+1]}\ !\ \texttt{Val} \,.\, \texttt{end}$$

where `self` denotes a distinguished parameter to abstractly represent the rank of a concrete Worker, set at run-time (i.e., $L_{\text{Pipe},\texttt{W[2]}}$ on page 14 is obtained by setting $\texttt{self} = 2$).

Inferring role variants (1). Our language of global types does not feature constructs to explicitly specify role variants. This is because they can be *automatically inferred* from intervals. Before formulating our inference procedure in full generality, we explain its key points with two examples.

Reconsider the Pipeline global type G_{Pipe} in Fig. 8. Suppose we aim to determine the behaviour of Worker a. The iteration domain in G_{Pipe} specifies two intervals: `1..k-1` and `2..k`.

- If a is contained in interval `1..k-1`, but it is not contained in interval `2..k`, then $a = \texttt{1}$. In this case, Worker a participates in exactly one iteration of the loop (i.e., the first one), as a sender.
- If a is not contained in interval `1..k-1`, but it is contained in interval `2..k`, then $a = \texttt{k}$. In this case, Worker a also participates in exactly one iteration of the loop (i.e., the last one), as a receiver.
- If a is contained in both intervals, then $\texttt{1} < a < \texttt{k}$. In this case, Worker a participates in *two* iterations of the loop: first as a receiver, and then as a sender.

The crucial insight demonstrated by this example is that the behaviour of any Worker is *completely* determined by the intervals that contain its rank; there are no other sources of behavioural variation. Moreover, since the number of intervals in any global type is bounded, the number of role variants is bounded as well: role `W` occurs with only two intervals in G_{Pipe}, so `W` has at most 2^2 variants.

Inferring role variants (2). Consider global type $G'_{\text{Pipe}} = \texttt{M} \rightarrow \texttt{W[1]} : \texttt{Init} \;.\; G_{\text{Pipe}}$, which prefixes G_{Pipe} with an initial communication from the Master to the first Worker. In this global type, role name `W` occurs actually with *three* intervals: two explicit ones in the iteration domain in G_{Pipe} (as before), and one implicit one in the initial communication, namely `1..1`. To see where this implicit interval comes from, note that the initial communication can be rewritten with `foreach`:

$$\texttt{foreach W\{i:1..1\} do (M} \rightarrow \texttt{W[i] : Init . cont) ;}\ G_{\text{Pipe}}$$

(Such rewriting is not always possible, because it generally does not preserve well-formedness; we do it here only to show what we mean with "implicit intervals".) Since role `W` occurs with three intervals in G'_{Pipe}, `W` has at most 2^3 variants. Four of these "potential variants" of `W` are *invalid*. For instance, there exists no rank a that is *both* in interval `1..1` *and* in interval `2..k`.

Inference procedure. We formulate our inference procedure as follows. Let $\mathsf{ival}(r, G)$ denote the set of intervals consisting of $\{D_j\}_{j \in J}$ for every $\texttt{foreach}\ R \cup \{r\}\{i_j : D_j\}_{j \in J}\ \texttt{do}\ G_1 \;;\; G_2$ in G and $E..E$ for every $r[E]$ in G. Note that ival does *not* interpret intervals into sets of concrete ranks; every element in $\mathsf{ival}(r, G)$ is syntactic, of the shape $E_1..E_2$. Every binary partition $\mathcal{D}, \bar{\mathcal{D}}$ of $\mathsf{ival}(r, G)$, of the total $2^{|\mathsf{ival}(r,G)|}$, characterises a potential variant of role r; we denote this variant as $r[\mathcal{D}, \bar{\mathcal{D}}]$. To check its validity, we construct a formula $\Phi(\mathcal{D}, \bar{\mathcal{D}})$. Let $k_1, k_2, \ldots$ denote the parameters in G.

$$\Phi(E_1..E_2) = E_1 \leq \texttt{self} \leq E_2 \qquad \Phi(\mathcal{D}, \bar{\mathcal{D}}) = \exists \texttt{self}.\Big[\Big[\textstyle\bigwedge_{D \in \mathcal{D}} \Phi(D)\Big] \land \Big[\textstyle\bigwedge_{\bar{D} \in \bar{\mathcal{D}}} \neg\Phi(\bar{D})\Big]\Big]$$

If $\exists k_1.\exists k_2.\ldots\Phi(\mathcal{D}, \bar{\mathcal{D}})$ is true, there exists at least one instantiation of parameters $k_1, k_2, \ldots$ such that there exists a individual (i.e., $\exists\texttt{self}$) whose rank is contained in all the intervals in $\mathcal{D}$ (i.e., $\bigwedge_{D \in \mathcal{D}} \Phi(D)$), and not contained in all the intervals in $\bar{\mathcal{D}}$ (i.e., $\bigwedge_{\bar{D} \in \bar{\mathcal{D}}} \neg\Phi(\bar{D})$). In more operational terms, if $\Phi(\mathcal{D}, \bar{\mathcal{D}})$ is true, there exists at least one run-time configuration of parameters in which at least one individual enacts the role variant characterised by $\Phi(\mathcal{D}, \bar{\mathcal{D}})$; thus, $r[\mathcal{D}, \bar{\mathcal{D}}]$ is valid. Conversely, if $\Phi(\mathcal{D}, \bar{\mathcal{D}})$ is false, there exists no such run-time configuration, meaning invalidity.

Thus, our inference procedure for variants of role r works as follows: (1) compute $\mathsf{ival}(r, G)$; (2) for every partition $\mathcal{D}, \bar{\mathcal{D}}$ of $\mathsf{ival}(r, G)$, check $\Phi(\mathcal{D}, \bar{\mathcal{D}})$; (3) $\Phi(\mathcal{D}, \bar{\mathcal{D}})$ is true iff $r[\mathcal{D}, \bar{\mathcal{D}}]$ is valid.

Inferring families. A *family* is a set of role variants that collectively constitute a consistent run-time configuration of an application. For instance, the Pipeline protocol has two families (Fig. 8): one for $\texttt{k} = \texttt{2}$ (front and last Worker), and one for $\texttt{k} > \texttt{2}$ (front, middle, and last Worker).

Role variant families can be inferred using a similar approach as for role variants. Let V_{all} denote the set of all inferred role variants. For every partition $V, \bar{V}$ of V_{all}, construct the following formula:

$$\Xi(V,\bar{V}) = \Big[\bigwedge_{r[\mathcal{D},\bar{\mathcal{D}}]\in V}\Phi(\mathcal{D},\bar{\mathcal{D}})\Big] \wedge \Big[\bigwedge_{r[\mathcal{D},\bar{\mathcal{D}}]\in\bar{V}}\neg\Phi(\mathcal{D},\bar{\mathcal{D}})\Big]$$

If $\exists k_1.\exists k_2.\ldots\Xi(V,\bar{V})$ is true, there exists at least one instantiation of parameters $k_1, k_2, \ldots$ such that only every variant in V is enacted by at least one individual, so V is a family.

4.4 Projection

Our final ingredient is a *projection* operator, $\upharpoonright$: it consumes as input a global type G and a role variant $r[\mathcal{D},\bar{\mathcal{D}}]$, and it produces as output *one* local type that specifies the behaviour of *all* individuals that enact $r[\mathcal{D},\bar{\mathcal{D}}]$. Below is an excerpt of the definition:

$$(r_1[x_1] \to r_2[x_2] : \{\ell_j\ .\ G_j\}_{j\in J}) \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}] = \begin{cases} r_2[x_2]\ !\ \{\ell_j\ .\ G_j \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}]\}_{j\in J} & \text{if } r_1 = r \neq r_2,\ x_1..x_1 \in \mathcal{D} \\ r_1[x_1]\ ?\ \{\ell_j\ .\ G_j \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}]\}_{j\in J} & \text{if } r_1 \neq r = r_2,\ x_2..x_2 \in \mathcal{D} \\ \bigsqcap\{G_j \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}]\}_{j\in J} & \text{if } r_1 \neq r \neq r_2 \end{cases}$$

$$(\mathtt{foreach}\ R\{i_j : D_j\}_{j\in J}\ \mathtt{do}\ G_1\ ;\ G_2) \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}] = \begin{cases} \ldots\ \textit{(omitted – see Supplement,[4] § II.3)} & \text{if } r \in R \\ \mathtt{foreach}\ R\{i_j : D_j\}_{j\in J}\ \mathtt{do} & \text{if } r \notin R \\ \quad (G_1 \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}])\ ;\ (G_2 \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}]) & \end{cases}$$

$$\mathtt{rec}\ X\ G \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}] = \mathtt{rec}\ X\ (G \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}]) \qquad X \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}] = X \qquad G \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}] = G \quad \textbf{if: } G \in \{\mathtt{cont}, \mathtt{end}\}$$

$\upharpoonright$ recursively traverses the structure of global type G and checks for every communication whether role variant $r[\mathcal{D},\bar{\mathcal{D}}]$ (i.e., an individual that enacts $r[\mathcal{D},\bar{\mathcal{D}}]$) participates as the sender or the receiver. If so, it adds a corresponding I/O action to the local type under construction; otherwise, it continues the traversal and *merges* projections of the continuations using $\sqcap$; the definition of $\sqcap$ is standard (e.g., [Deniélou et al. 2012]), extended in the natural way for `foreach`. As usual (e.g., [Coppo et al. 2016; Honda et al. 2016]), projection is partial: it is undefined for unsafe protocols.

If `foreach` is encountered, our projection operator checks if r is in the iteration domain. If it *is not*, $r[\mathcal{D},\bar{\mathcal{D}}]$ participates in *all* iterations of the loop (i.e., `foreach` must be preserved in the local type under construction), and in every iteration, it behaves according to the projected body (possibly empty, i.e., `cont`). Otherwise, if r *is* in the iteration domain, $r[\mathcal{D},\bar{\mathcal{D}}]$ participates in only *some* iterations (i.e., `foreach` must not be preserved), for which special measures need to be taken, represented above as "…"; see the Supplement,[4] § II.3 for the full definition.

Example 4.9. Role `S` *does not* occur in the iteration domain of `foreach` in G_{Pget}, Ex. 4.3, so must be preserved in $G_{\text{Pget}} \upharpoonright \mathtt{S}[\{0..0\},\emptyset]$. This is as expected: Server receives from all Fetchers, so it participates in all iterations. In contrast, role `F` *does* occur in the iteration domain, so `foreach` is lost in $G_{\text{Pget}} \upharpoonright \mathtt{F}[\{2..k\},\{1..1\}]$. This, too, is as expected: every Fetcher sends exactly once to Server.

4.5 Decidability and Correctness

Inference procedures. We first address the decidability of our inference procedures in § 4.3.

THEOREM 4.10. *Inference of role variants and families is decidable.*

PROOF. Because $\mathsf{ival}(r,G)$ is finite (i.e., the set of intervals that occur syntactically in G), the number of binary partitions $\mathcal{D},\bar{\mathcal{D}}$ is finite as well. Also, $\Phi(\mathcal{D},\bar{\mathcal{D}})$ and $\Xi(V,\bar{V})$ are formulas over $\langle\mathbb{A},+,0,\leq\rangle$, which is decidable (see § 4.1 and Ex. 4.1, 4.2). □

Well-formedness. We guarantee correctness and safety for *well-formed* global types. Let $\mathbb{K} \rightharpoonup \mathbb{A}$ denote the set of all partial *substitutions* of values for parameters, ranged over by σ,τ; let $G\langle\!\langle\sigma\rangle\!\rangle$ denote the *instantiation* of G in accordance with σ. A substitution σ *closes* global type G if $G\langle\!\langle\sigma\rangle\!\rangle$ has no parameters; $G\langle\!\langle\sigma\rangle\!\rangle$ is *well-closed* if all intervals in $G\langle\!\langle\sigma\rangle\!\rangle$ are non-empty.

A global type G is well-formed if for all σ such that $G\langle\!\langle\sigma\rangle\!\rangle$ is well-closed: (1) index variables and type variables in G are bound by `foreach` and `rec`; (2) `rec` does not occur under `foreach` in G; (3) an "inner" `foreach` in G cannot range over role names already ranged over by an "outer" `foreach`; (4) all intervals in the same iteration domain in $G\langle\!\langle\sigma\rangle\!\rangle$ have the same length. Condition (2) ensures that every iteration of a loop terminates; we support only tail recursion. Condition (3) ensures that the number of iterations an individual participates in can be computed statically. Condition (4) ensures that the "table" for every iteration domain (e.g., Fig. 8) has a well-defined number of "rows".

Theorem 4.11. *Checking well-formedness is decidable.*

Proof. Conditions (1), (2), and (3) are structural and independent of σ; checking them is trivially decidable. In contrast, checking condition (4) requires universal quantification over the set $\{\sigma \mid G\langle\!\langle\sigma\rangle\!\rangle \text{ is well-closed}\}$, which can be infinite. To check (4), we construct a first-order formula over $\langle\mathbb{A},+,0,\leq\rangle$, which is decidable (see § 4.1 and Ex. 4.1, 4.2), as follows. Let $k_1,k_2,\ldots$ denote the parameters that occur in G, and let $\mathcal{I}$ denote the set of all iteration domains that occur in G:

$$\Psi_{\neg\emptyset}(\{i_j : E_{j,1} .. E_{j,2}\}_{j\in J}) = \textstyle\bigwedge_{j\in J} E_{j,1} \leq E_{j,2} \qquad \Psi(\mathcal{I}) = \forall k_1.\forall k_2.\ldots\left[\textstyle\bigwedge_{I\in\mathcal{I}} \Psi_{\neg\emptyset}(I) \Rightarrow \textstyle\bigwedge_{I\in\mathcal{I}} \Psi_{=}(I)\right]$$

$$\Psi_{=}(\{i_j : E_{j,1} .. E_{j,2}\}_{j\in J}) = \textstyle\bigwedge_{j_1,j_2\in J}(E_{j_1,2}-E_{j_1,1} = E_{j_2,2}-E_{j_2,1})$$

Now, $\Psi(\mathcal{I})$ is true iff (4) holds. □

Correctness and safety. In words, ***correctness*** of $\upharpoonright$ means that the behaviour specified by an instantiated well-formed global type G equals the joint behaviour specified by the instantiated local types projected from G, namely one for every individual that enacts an inferred role variant.

Let $L\langle\!\langle\tau\rangle\!\rangle$, $\mathcal{D}\langle\!\langle\tau\rangle\!\rangle$, and $\bar{\mathcal{D}}\langle\!\langle\tau\rangle\!\rangle$ denote the instantiation of the parameters in local type L and sets of intervals $\mathcal{D},\bar{\mathcal{D}}$ according to τ (cf. $G\langle\!\langle\sigma\rangle\!\rangle$, above). Let $\equiv$ denote trace equivalence of the LTSs induced by a global type and a system (parallel composition) of local types [Deniélou and Yoshida 2013]. The following theorem states correctness; see the Supplement,[4] § II.4 for our proof.

Theorem 4.12. *For all well-formed G and σ such that $G\langle\!\langle\sigma\rangle\!\rangle$ is well-closed:*

$$G\langle\!\langle\sigma\rangle\!\rangle \equiv \{(G \upharpoonright r[\mathcal{D},\bar{\mathcal{D}}])\langle\!\langle\tau\rangle\!\rangle \mid \exists a : \mathcal{D} \uplus \bar{\mathcal{D}} = \mathsf{ival}(r,G),\ \tau = \sigma \cup \{\texttt{self} \mapsto a\},\ \models \Phi(\mathcal{D}\langle\!\langle\tau\rangle\!\rangle, \bar{\mathcal{D}}\langle\!\langle\tau\rangle\!\rangle)\}$$

Projection guarantees ***safety*** if the joint behaviour specified by the instantiated local types projected from a well-formed global type is free of deadlocks and reception errors. Safety is a direct consequence of correctness: deadlocks and reception errors cannot be specified in our language of global types, so a correct projection never produces an unsafe system of local types. The formalisation of the following corollary relies on the same LTS semantics of global types and systems of local types as the one that underlies Thm. 4.12; see [Deniélou and Yoshida 2013].

Corollary 4.13. *Projection guarantees safety.*

5 IMPLEMENTATION

5.1 Extension of Scribble based on Distributed, Role-Parametric MPST

We extend the Scribble protocol language for role-parametric protocols based on our core formalism in § 4 and the syntactic sugars outlined in § 4.2. Our presented design results from experimenting with various combinations of primitives and communication patterns for a range of examples (summarised in Fig. 15). Fig. 11 (top) outlines our grammar: we add `foreach`, and cover the special global type arrows from Fig. 10 by extending the global interaction of Scribble (`from`/`to`) with indexed roles p and inline message choices ℓ_1 `or` ... `or` ℓ_n, and adding the `pair`/`pipe` primitives; for simplicity, we show a restriction to one-dimensional indices (Ex. 4.1). g is a protocol name, and A stands for basic boolean expressions for constraints on index variables; other notation not explicitly defined here (e.g., r, E) is as in § 4.2. p^1 means the restriction of D to `[`E`,` E`]` or `[`i`]`. In our experience, these particular primitives are beneficial for writing protocols and using the generated APIs (cf. "manual" `foreach` encodings), and also run-time performance.

Fig. 11 (bottom) illustrates the correspondence between our formal notation and Scribble syntax. The Scribble PP choice subsumes the case of unary choices. For paired/pipelined-PP (top right), `pair` corresponds to $\overset{*}{\Rightarrow}$, where the choice is made *independently* by each $r_1[i]$ to its opposing peer in the r_2 interval; `pipe` may be used here as the special case where $r_1 = r_2$ (so the choice is made

$$
\begin{array}{lll}
P & ::= & \texttt{global protocol}\ g\ \texttt{@}A\,(\texttt{role}\ r_1,\ \ldots,\ \texttt{role}\ r_n)\ \{\ G\ \} \qquad p\ ::=\ r\texttt{[}y\texttt{]} \qquad y\ ::=\ D \mid i \qquad D\ ::=\ E_1,\ E_2 \\
G & ::= & \ell_1\ \texttt{or}\ \ldots\ \texttt{or}\ \ell_n\ \texttt{from}\ p_1\ \texttt{to}\ p_2\texttt{;} \mid \texttt{choice at}\ p^1\ \{\ G_1\ \}\ \texttt{or}\ \ldots\ \texttt{or}\ \{\ G_n\ \} \mid \texttt{do}\ g(r_1,\ \ldots,\ r_n)\texttt{;} \mid G_1\ G_2 \\
 & \mid & \ell\ \texttt{pair}\ p_1\ \texttt{to}\ p_2\texttt{;} \mid \ell\ \texttt{pipe}\ r\texttt{[}D\texttt{];} \mid \texttt{foreach}\ r_1\texttt{[}i_1\!:\!D_1\texttt{]},\ \ldots,\ r_n\texttt{[}i_n\!:\!D_n\texttt{]}\ \{\ G\ \}
\end{array}
$$

Role-parametric subprotocols	$\texttt{foreach}\ R\{i_j\!:\!D_j\}_{j\in J}\ \texttt{do}\ G_1$	$\texttt{foreach}\ r_1\texttt{[}D_1\texttt{]},\ \ldots,\ r_n\texttt{[}D_n\texttt{]}\{\ G\ \}$
Choice(s)	Scatter/gather/all-to-all: $\rightarrow$	Paired/pipelined unicasts: $\Rightarrow$
Peer-to-peer (PP): $*$	$r_1[D_1] \xrightarrow{*} r_2[D_2]\{\ell_1, \ell_2\}\ ;\ G$ $\ell_1\ \texttt{or}\ \ell_2\ \texttt{from}\ r_1\texttt{[}D_1\texttt{]}\ \texttt{to}\ r_2\texttt{[}D_2\texttt{]};\ G$	$r_1[D_1] \xRightarrow{*} r_2[D_2]\{\ell_1, \ell_2\}\ ;\ G$ $\ell_1\ \texttt{or}\ \ell_2\ \texttt{pair}\ r_1\texttt{[}D_1\texttt{]}\ \texttt{to}\ r_2\texttt{[}D_2\texttt{]};\ G$
Master-slaves (MS): 1	$p^1 \xrightarrow{1} r_2[D_2]\{\ell_1.G_1, \ell_2.G_2\}$ $\texttt{choice at}\ p^1\ \{\ \ell_1\ \texttt{from}\ p^1\ \texttt{to}\ r_2\texttt{[}D\texttt{]};\ G_1\ \}$ $\texttt{or}\ \{\ \ell_2\ \texttt{from}\ p^1\ \texttt{to}\ r_2\texttt{[}D\texttt{]};\ G_2\ \}$	$r \xRightarrow{1} E_1 \ldots E_2\{\ell_1.G_1, \ell_2.G_2\}$ $\texttt{choice at}\ r\texttt{[}E_1\texttt{]}\ \{\ \ell_1\ \texttt{pipe}\ r\texttt{[}E_1,E_2\texttt{]};\ G_1\ \}$ $\texttt{or}\ \{\ \ell_2\ \texttt{pipe}\ r\texttt{[}E_1,E_2\texttt{]};\ G_2\ \}$

Fig. 11. Practical syntax for role-parametric protocols: (top) extended Scribble grammar; (bottom) illustration of global type and Scribble correspondence (cf., the formal syntactic sugars in Fig. 10).

at each step along the interval). The MS choice is for more than one case (we show only binary choices for brevity). For pipelined-MS (bottom right), the interaction must be a `pipe`, where the choice is *propagated* along the interval, for the MS choice to be consistent at all receivers. In the Scribble `foreach`, $(r\texttt{[D]})_{1..n}$ enumerates the ranges used in the formal notation (cf. Rem. 1).

We omit the implementation details of basic syntactic checks (e.g., valid combinations of `choice` and `from`/`pair`/`pipe` as per Fig. 11) and well-formedness (§ 4.5) that are as expected. Cond. (4) of well-formedness, valid role variants, and variant families are similarly determined following § 4.3. Our toolchain integrates Scribble with Z3 to check the induced constraints; e.g., for Pipeline (Fig. 8), this generated Z3 assertion confirms the middleman is a valid variant:

```
(assert (exists ((self Int) (K Int)) (and (> K 1) // Annotated domain constraint
   (>= self 1) (<= self (- K 1)) (<= 1 (- K 1)) (>= self 2) (<= self K) (<= 2 K) // 𝒟 constraints for W2..K-1
   (not (and (>= self 1) (<= self 1))) (not (and (>= self K) (<= self K)))  )))  // 𝒟̄ constraints for W2..K-1
```

5.2 Communicating FSM Based Representation of Local Types

Our toolchain uses an internal representation of local types (§ 4.2) based on communicating FSMs with *gather/scatter I/O* and *parameterised nesting* of sub-FSMs within states. The correspondence between the syntactic types and our FSMs is straightforward: we outline the correspondence below, and provide a full definition in the Supplement,[4] § III.1.

Based on our local types, we write $r[D]\dagger\ell$, $\dagger \in \{!, ?\}$, for the scatter/gather I/O of our FSMs. Fig. 12 shows the FSMs for MS and PP choices; the latter demonstrates the basic FSM for `foreach`.

MS The $!^1$ send-to-all local type (§ 4.2), which selects the *same* choice at all peers, corresponds to an FSM scatter: the type $r[y] \ddagger \{\ell_j\ .\ L_j\}_{j\in J}$ is simply represented as a state with each of the J cases as a separate transition. Dually, MS input is implicitly a standard ? from a single peer (i.e., there is no $?^1$), for MS choices to be consistent across all receivers.

PP Non-unary PP choices are represented as nested FSMs, via $\texttt{foreach}$ desugaring of $\xrightarrow{*}$ / $\xRightarrow{*}$ (Fig. 10). For the example type $\texttt{foreach}\ r\{i\!:\!D\}\ \texttt{do}\ r[i]\ \dagger\ \{\ell_j\ .\ \texttt{cont}\}_{j\in J}; L$, the subprotocol – i.e., a $\dagger$-choice of J cases – is nested and parameterised within the initial state of the FSM for the continuation L, denoted s_0^L. This FSM is just a representation of the local type behaviour: first repeat the nested FSM for each value of i in D in sequence, then perform one of $\alpha_{1..n}$ (standard state transition). At the local type level, unary PP choices coincide with MS choices: as an optimisation, we represent unary PP choices similarly to MS choices (i.e., without nesting).

We introduce some notation for our FSMs, that we shall use for defining our Go API generation. A *role variant FSM* (henceforth, *FSM*) is a tuple $M = (\mathbb{S}, \mathbb{R}, s_0, \mathbb{T}, \delta, \phi)$. Apart from the last element, all are standard [Deniélou and Yoshida 2012]. $\mathbb{S} = \{s_1, s_2, \ldots\}$ is a set of *state* identifiers; $s_0 \in \mathbb{S}$ is

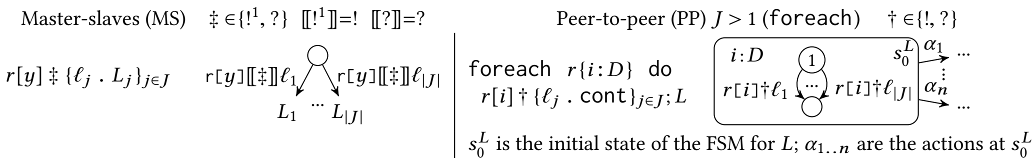


Fig. 12. Representation of local types as communicating FSMs (partial illustration).

the *initial state*. $\mathbb{R} = \{r_1, r_2, \ldots\}$ is a set of role names. $\mathbb{T} = \{\ell_1, \ell_2, \ldots\}$ is a set of *message labels*. $\delta : \mathbb{S} \times \{\alpha_1, \alpha_2, \ldots\} \to \mathbb{S}$ is the *transition function*, where $\alpha_1, \alpha_2, \ldots$ are *local actions* of the form $r[y] \dagger \ell$ with $\dagger \in \{!, ?\}$. Finally, $\phi : \mathbb{S} \to \mathbb{M} \times \mathbb{P}$ is the *nesting function*, where $\mathbb{M}$ is the domain of FSMs and $\mathbb{P}$ is the domain of sets of indexed intervals, ranged over by P.

By the syntax and properties of global types and projection (§ 4), every transition from a state has the same action kind $\dagger$; and every transition of an *input* state has the same *peer* $r[y]$. We also collapse every occurrence of `end` in a local type, if any, to a single terminal state at the top level of its FSM (similarly, for `cont` at the top level of a `foreach`). A "plain" state (i.e., that we depict without a nested FSM) corresponds to a state that nests a sole (terminal) state.

5.3 State Channel API Types Generation for Go

We first explain the key types and methods generation for states, I/O and branching, and nested FSMs. We simplify the presentation in two ways. First, we abstract from the details of specific naming schemes for types and methods: we use the notation $[\![\cdot]\!]$ to stand for any concrete name mapping, e.g., $[\![s]\!]$ is a Go type name for a state s, and $[\![s, \alpha]\!]$ is an I/O method name for action α from s. Second, we assume a "flat" naming scheme for methods, instead of the scheme presented in § 3, e.g., `S_Send_Head(...)` instead of `S.Send.Head(...)`; we illustrate the more cosmetically elaborate types generation for the latter in the Supplement,[4] § I.1.1. As noted earlier, a local program may use Go package/type aliases, or the user could supply custom names as Scribble annotations.

In the following, assume a variant $v = r[\mathcal{D}, \bar{\mathcal{D}}]$ (of some protocol g), and let s be a state in the FSM of v such that $\delta(s) = \{\alpha_j \mapsto s_j\}_{j\in J}$ and $\phi(s) = M, P$. Let $[\![M_0]\!]$ (resp. $[\![M_{\mathsf{End}}]\!]$) be a type name derived from the initial (resp. terminal, if present) state of M; and $[\![v]\!]$ be the type name of the Endpoint for v (e.g., the type of `M` on the left of Fig. 5, line 3).

Nested FSMs. Fig. 13 (top) shows the types generated w.r.t. the nesting of M in s. $[\![s]\!]$ is the "main" (or entry) *state channel* type for s (e.g., the initial state, or result of the previous I/O method). It offers a `Foreach` method, that takes a function from $[\![M_0]\!]$ to $[\![M_{\mathsf{End}}]\!]$, i.e., an implementation of the nested behaviour. The result (after all nested behaviours are completed) is $[\![s]\!]'$, an intermediary type for finally performing a transition out of s. The basic Go code for executing a nested FSM and the subsequent state transition may thus look like:

`s.Foreach(nested).m(...)` `s` is a variable of type $[\![s]\!]$, `nested` is of type `func(`$[\![M_0]\!]$`)` $[\![M_{\mathsf{End}}]\!]$

`Foreach` is generated to repeat `nested` over the intervals P embedded into the API, and `m` is the I/O method generated for the subsequent transition, explained next.

I/O and branching. The generation of I/O methods depends on which kind of state s is.

<u>Output or unary-input</u> For each $\alpha_j = r_1[y] \,!\ell_j$ with $|J| > 1$, or for $\alpha_1 = r_j[y] \,?\ell_1$ when $|J| = 1$:

`func (c *`$[\![s]\!]'$`)` $[\![s, \alpha_j]\!]$ `(m` $\langle\ell_j\rangle$`) *`$[\![\delta(s, \ell_j)]\!]$ `{ ... }` $\langle\ell_j\rangle$ is, e.g., $*\ell_j$ for `Send/Receive`, `[]`ℓ_j for `Scatter/Gather`

($c *[\![s]\!]'$) is the method receiver (i.e., the intermediary result type of `Foreach`), $[\![s, \alpha_j]\!]$ is the method name, and $\langle\ell_j\rangle$ stands for the parameters according to the I/O action kind (e.g., singleton `Send` `/Receive` are special cases of `Scatter/Gather`). We omit details of further variations, e.g., `Reduce`.

```
type ⟦s⟧ struct { Err error; id uint64; ep *⟦v⟧; ... } // State channel type: first do Foreach
func new_⟦s⟧(...) *⟦s⟧ { ... return &⟦s⟧{...} }        // Private constructor (used internally within API)
type ⟦s⟧′ struct { Err error; id uint64; ep *⟦v⟧; ... } // Intermediary type (after Foreach done)
func (c *⟦s⟧) Foreach(nested func(int, *⟦M0⟧) ⟦MEnd⟧) *⟦s⟧′ { ... } // int is the Foreach index param
```

State type		Method name, signature		
State	Peer	I/O	Label, value	Succ.
W_2toK_1	W_self_sub1	Branch()		*W_2toK_1_Cases
W_2toK_2	W_self_plus1	Send	Next(a *Next)	*W_2toK_1
W_2toK_3	W_self_plus1	Send	Done(a *Done)	End
W_2toK_1_Cases is an interface, implemented by the below case types				
$t_{\texttt{Next}}$	n/a	Receive	Next(a *Next)	*W_2toK_2
$t_{\texttt{Done}}$	n/a	Receive	Done(a *Done)	*W_2toK_3

```
var n next; var d Done
for {
  switch c := w.W_self_sub1.Branch().(type) {
  case *t_Next:
    w = c.           Receive.Next(&n).
          W_self_plus1.Send.    Next(&n)
  case *t_Done:
    return c.        Receive.Done(&d).
          W_self_plus1.Send.    Done(&d)    } }
```

Fig. 13. State channel API generation: (top) state channel and Foreach type signatures; (bottom) type switch branch API types and I/O methods for the $\texttt{W}_{2..\texttt{K}-1}$ variant in Ring (Ex. 4.4), and an example implementation.

<u>Branch-input ($|J| > 1$)</u> We show the branch API generation that targets Go *type switch* statements.

```
type ⟦s⟧_Cases interface { ⟦s⟧_Case() }
func (c *⟦s⟧′) Branch() *⟦s⟧_Cases { ... }
```

```
type ⟦s⟧_ℓj struct { Err error; id uint64; ep *⟦v⟧; ... }
func (*⟦s⟧_ℓj) ⟦s⟧_Case() { } // Implement _Cases i/face
func (c *⟦s⟧_ℓj) ⟦s, αj⟧(m *ℓj) *⟦δ(s, ℓj)⟧ { ... }
```

On the left, $[\![s]\!]$_Cases is an interface representing the valid choice cases: on the right, for each $\alpha_j = r[x]?\ell_j$, we generate an $[\![s]\!]_{\ell_j}$ type that implements this interface (via the token $[\![s]\!]$_Case method) and offers an appropriate $[\![s, \alpha_j]\!]$ input method. The Branch method, with receiver $*[\![s]\!]'$ (like the I/O methods above), is then generated to block until a message is received, and return the corresponding implementation of the $[\![s]\!]$_Cases interface.

As an example, Fig. 13 (bottom) summarises the branch API types generated for the $\texttt{W}_{2..\texttt{K}-1}$ "middleman" variant in Ring (Ex. 4.4) and gives a user implementation. A type switch `switch c := ....(type)` evaluates the expression (assigned to c) and selects the first case that matches the run-time *type* of the result. IDEs can auto-generate exhaustive `switch` cases for the programmer.

Our implementation simplifies the generated API as expected in certain cases. E.g., when M is a single state (i.e., s is a "plain" state), the API generation skips the intermediary $[\![s]\!]'$ type and Foreach method, and sets the receiver of the I/O methods directly to $[\![s]\!]$. Our examples in this paper assume a $[\![\cdot]\!]$ that maps terminal states to an End type; we also set the result of terminal I/O methods to a *non*-pointer End type for stronger safety, as it prevents, e.g., `return nil`.

Note that FSMs are explicitly used only at *compile*-time for the presented types generation: the point of the types is to statically guide the FSM structure implicitly in the program. At run-time, the only checks introduced by our APIs are on session initiation parameters and channel linearity, as explained in the next paragraph.

Automated inlining of dynamic checks. The static assurances of the generated Go API types are supported by automated inlining of a few kinds of lightweight run-time checks into the API.

Go preliminaries: a *defer* statement pushes a function call (e.g., a channel closure) onto a list; the list is executed after the surrounding function returns. *Panic* is a built-in function that stops the control flow of the calling goroutine and executes any deferred calls at each level of its call stack; control flow may be regained by (a deferred call to) the built-in *recover* function.

<u>Endpoint initiation</u> The first check is on the parameter values supplied to the Endpoint constructor (e.g., K in Fig. 5, line 3), derived straightforwardly from the $\mathcal{D}, \bar{\mathcal{D}}$ elements of the variant. This is a simpler version of the compile-time Z3 assertion illustrated in § 5.1 that just checks the constraints on concrete values (as a Go expression) rather than existential quantifications.

Secondly, the `Dial/Accept` methods are generated to check for, e.g., duplicate connections; similarly, the top-level `run` checks for missing connections. A violation of these checks raises a panic.

State channel API The implicit usage contract of a state channel API is to use every channel *instance* exactly once, i.e., *linearly*. Repeat usage is dynamically checked by assigning a fresh ID value to each channel instance (the `uint64` fields in Fig. 13, top) and recording for each Endpoint the ID of the currently active channel: every I/O method is generated to check the target channel is the indeed the currently active one. Endpoint completion, guided by the `End` return type of the generated top-level `run` method, is an (at most) one-time deferred check within `run`.

Error handling and failures. We integrate the call-chaining nature of the presented APIs with the explicit error handling paradigm of Go. The API is generated to (1) set the state channel `Err` field (Fig. 13, top) in the successor channel instance if the preceding action caused an error (`error` is an in-built interface type), or else `Err` is `nil`; and (2) raise a panic when an I/O method is called on a channel whose `Err` is not `nil`. By our safety guarantees (see below), an error means a failure in the underlying I/O or networking facilities, or perhaps the reception of an incorrect message type when interacting with a potentially unsafe participant—the deserialization operations in our generated API code for inputs serve as implicit compliance checks on received message types.

Idiomatic Go error handling using a state channel API is as below (cf. lines 18–19 in Fig. 5).

```
if m3 := m2.F_1toK.Scatter.Job(split(&meta)); m3.Err != nil { // Explicit handling (e.g., networking failure)
  ... } else { ... m4 := m3.F_1toK.Reduce.Data(&data, agg) ... } // Using m3 with m3.Err != nil would raise a panic
```

Here, we use the standard Go construct `if x := f(); P(x) { g(x) } else { h(x) }`, which first evaluates `f()` and assigns the result to `x`, then evaluates `P(x)` to true or false, and finally executes `g(x)` or `h(x)`; the scope of `x` is constrained to this statement. The above code thus first attempts a scatter to the Fetchers. If no error (e.g., network failure) occurs, `m3` is the expected successor state channel, `m3.Err` is nil, and the then-branch is executed; if an error occurs, `m3.Err` is non-nil and the else-branch is executed. Handling errors in this way is idiomatic Go.

5.4 Practical Safety Guarantees of our Generated APIs

Our results in § 4.5 ensure, for a given family in a well-formed role-parametric protocol, the set of projections onto each variant constitutes a safe, distributed decomposition of the protocol. In other words, a distributed instance of this protocol (i.e., a *session*) is guaranteed free from reception errors, deadlocks and orphan messages, at the level of abstraction of our target model of asynchronous, pairwise-ordered and reliable message passing between the endpoints. The purpose of the API generation step of our framework is then to promote *compliance* of concrete endpoint implementations to their projections via native Go type checking, supported by the dynamic checks built into the API (§ 5.3).

Specifically, a generated state channel API ensures: in a successfully initiated session,

a statically *well-typed endpoint implementation will* never *perform a non-compliant I/O action w.r.t. the run-time instantiation of the role-parametric protocol, up to premature session termination.*

This is because the *only* way to *attempt* a non-compliant I/O action is to violate linear usage of a channel instance, in which case the in-built API check will (by default) raise a panic *without* actually performing the offending action. Such a situation effectively results, at worst, in an incomplete or premature termination of the endpoint, and thus the session, w.r.t. the protocol. Note, however, that premature termination is always a caveat in practice, due to program errors outside the session code, or node/network failures. In this regard, our API generation considers linearity violations and failures (via `Err`) uniformly, appealing to Go's in-built `panic`, `defer` and `recover` facilities.

Once a session is initiated, the only dynamic checks are on linear channel usage, giving an affine form [Tov and Pucella 2011] of the *MPST safety* discussed above. If the simple linearity condition of our APIs is respected, however, Go type checking is sufficient to ensure MPST safety. It would thus be possible to combine our approach with a technique for statically checking linear resource usage, given such a technique (with associated restrictions), to obtain the classical MPST safety outright.

Another highlight of our approach, and a basis for safety, is that the API generation *internalises* the management of parameter values and index expressions related to identifying the session peer(s) of every I/O action in the protocol—the user-supplied arguments of the generated I/O methods relate only to messages. As observed by Samofalov et al. [2005] of process rank indices/expression bugs in the setting of MPI programming, incorrect management of indices and parameters can be a tricky source of communication errors in practice.

On static channel linearity. We note *dynamic linearity checks are* **not** *fundamental to our overall approach*. By our results in § 4.5, our framework is amenable to the use of alternative API generation methods for separate endpoints: our toolchain also supports *callback*-based API generation, illustrated below for the first two states of `M` in Pget (Fig. 4):

```
M.register(M_1.state, func(c Cache, meta *Meta) { c.meta = meta }) // Callback for M_1: F_1?Meta
 .register(M_2.state, func(c Cache) { new M_2.F_1toK.Job(split(c.meta)) }) // M_2: F[1,K]!Job
 .... // Callbacks registered by user for each state on the generated Endpoint M
```

The above style of generated API encapsulates all communication channels under the API and internalises the *FSM* itself: after session initiation, the API calls back the user-supplied, state-specific functions at each state (upon message receipt for input states). Consequently, a Go endpoint program using the callback API enjoys *fully static* MPST safety (for a successfully initiated session with compliant peers); the tradeoff is requiring programming in an *event-driven* style.

The main API style presented in this paper promotes session programming in Go that is close to standard channel/socket based APIs (and the session π-calculi in MPST formalisms). One advantage is it allows us to re-implement existing Go programs more directly, as part of evaluating the applicability of our framework (see § 6). In our experience, debugging *local* linearity violations (as exceptions) is much simpler than the full task of debugging reception errors or deadlocks between distributed, non-compliant endpoint implementations.

The interested reader may find details on the Scribble-Go Runtime in the Supplement,[4] § III.2.

6 EVALUATION

We evaluate our framework in terms of run-time performance (§ 6.1), and applications (§ 6.2), using a machine with an Intel i7-8770 processor (6 physical and 6 virtual cores) and 16GB RAM, running Debian 9.1 and Go version `go1.11.2`. We used the Go benchmarking tools (https://godoc.org/testing).

6.1 Run-time Overheads of Generated APIs

Microbenchmarks. We measure the overheads introduced by our framework during session execution, due to using the generated state channel API, our Runtime, and dynamic linearity checks. We first present microbenchmarks as a worst-case for the above overheads in isolation, by performing no work other than I/O. We use three kinds of microbenchmark programs, for the core patterns: **One-to-Many** (multi-destination send, single-source receive), **Many-to-One** (single-destination send, multi-source receive), and **Many-to-Many** (multi-destination send, multi-source receive). Each benchmark kind is parameterised on a k: in the first two, k is the number of goroutines at the Many side; in the third, k is divided evenly between sender/receiver goroutines.

We implement each benchmark by two methods. **(1) Scribble-Go**: we specify the above patterns as protocols in our extended Scribble and implement the Endpoints using our generated APIs. For each Endpoint, we have two versions of initiation that differ only by the selected Runtime transport, **`shm`** or **`tcp`**. **(2) Go base cases**: each Scribble-Go program has a Go base case that corresponds to

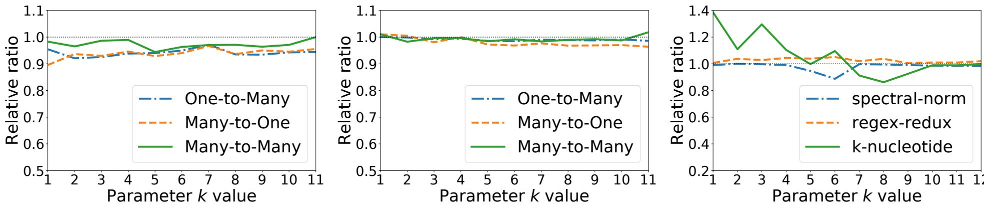


Fig. 14. **Scribble-Go** exec. time vs. **Go base cases**: (left) shm micro, (middle) tcp micro, and (right) CLBG.

replacing all occurrences and uses of state channels by direct references and uses of the underlying communication facilities, i.e., (unbounded) Go channels, or TCP sockets from the `net` package. We specify messages as having an `int` payload, and let k range over 1..11.

We measure the *execution time* from session start at the first sender (after *all* goroutines and connections established – in Scribble-Go, that is after entering the generated top-level `run`), to the end at the last receiver (before *any* connections closed). Since the execution time of a single instance of the above patterns is very small (on the order of nanoseconds), we repeat the communication actions (i.e., extend the "session length") in a loop of N iterations in each endpoint program and take the mean (N is set by the benchmarking tool, e.g., $> 10^6$, such that a run exceeds one second). The `tcp` endpoints are run as intra-process goroutines by the same setup as for `shm`, communicating through localhost TCP. We repeat each benchmark run 40 times and take the mean.

Fig. 14 (left) shows Go base case `shm` session execution time relative to Scribble-Go: x ranges over the value of k, and y is given by t_{go} / t_{api} ($y = 1$ is the baseline). The relative overheads of Scribble-Go are ~10% in most cases, over the range of k; for reference, we note that the *absolute* overhead per pattern is ~20 nanoseconds. Fig. 14 (middle) shows the corresponding results for `tcp`: the overheads are mostly < 3%. We remark that the relative overheads will continue to diminish as latency increases, e.g., for TCP over LAN or the Internet.

Case study: Computer Language Benchmarks Game (CLBG). We next present benchmarks using existing applications from Debian's CLBG [Gouy 2017], a repository of programs used to compare the performance of different languages (e.g., [Brunthaler 2010; Shirako et al. 2009; St-Amour et al. 2012; Wrigstad et al. 2010]). We use three concurrent Go programs: **(a)** `k-nucleotide` counts occurrences of molecule sequences in a DNA string, **(b)** `regex-redux` matches regex patterns against a DNA string, and **(c)** `spectral-norm` computes the greatest eigenvalue of a matrix. All three are based on scatter/gather work parallelisation between goroutines using Go channels. We take the original programs, written by the Go Authors, as the **Go base cases**. For **Scribble-Go**, we specify the (previously implicit) application protocols using our extended Scribble, each parameterised on a number $1 \leq k \leq 12$ of "worker" goroutines; and modify the original programs by replacing all vanilla Go channels, sends and receives with `shm` state channels and calls to the generated APIs.

For these macrobenchmarks, we measure the execution time of the *whole* application (i.e., including channel creations, Scribble-Go Endpoint initiations, etc.). We use the standard inputs defined in the CLBG, and take the mean of 20 repetitions for each application. Fig. 14 (right) shows the execution time of the Go base cases relative to Scribble-Go: x ranges over k, and y is t_{go} / t_{api}.

The results show Scribble-Go performs at least as well as the original programs in most cases; we expect the cost of computations in real applications such as these will often render the overheads negligible, considering the absolute values measured in the microbenchmarks. Scribble-Go is actually faster in some cases for `regex-redux` and `k-nucleotide` (observed for different versions of our Runtime). We believe this is due to including channel creations in the time measurement, and a

		Pt	Sc	Ga	FE	Pipe	MS	PP	Rec	Del
Core I/O patterns	1. One-to-Many (§ 6.1)		●		○					
	2. Many-to-One (§ 6.1)			●	○					
	3. Many-to-Many (§ 6.1)		●	●	○					
	Above, ○ are possible alt. implementations									
Parallel Topologies	4. Pipeline (§ 4)	●				●				
	5. Ring (§ 3; 4)	●			●	●		●	●	
	6. Hadamard (§ 4)	●			●					
	7. Mesh (§ 4)	●			●	●				
	8. Fork-Join		●	●						
Applications	9. Pget[2] (□ is the difference between the two versions in § 3.2; § 3.3)	●	●	●	□					●
	10. Vickrey auction (Supplement, § IV.1.2)	●	●	●	●		●		●	
	11. Jacobi solution of discrete Poisson equation. [Bejleri et al. 2009]	●	●	●	●		●		●	
	12. n-body simulation (based on Ring) [Bejleri et al. 2009]	●			●	●		●	●	
	13. Iterative linear equation solver (based on Mesh) [Ng and Yoshida 2015]	●	●		●	●	●		●	
	14. `k-nucleotide` [Gouy 2017] (§ 6.1)		●	●						
	15. `regex-redux` [Gouy 2017] (§ 6.1)		●	●						
	16. `spectral-norm` [Gouy 2017] (§ 6.1)		●	●			●		●	
	17. Fibonacci [Lange et al. 2017]	●			●					
	18. Quote-Request [Austin et al. 2004; Ng and Yoshida 2015]		●	●	●		●		●	
	19. P2P multiplayer game [Scalas et al. 2017]	●			●	●		●	●	●
	20. Web Crawler [Akhmadeev 2016; Neykova and Yoshida 2017]	●	●	●	●					
	21. n-buyers [Coppo et al. 2016; Honda et al. 2016]	●			●		●		●	

Pt: point-to-point; Sc: Scatter; Ga: Gather; FE: Foreach; Pipe: Pipeline; MS: MS choices; PP: PP choices; Rec: Recursion; Del: Delegation

Fig. 15. Role-parametric protocols for communication patterns, topologies and applications in Scribble-Go.

small restructuring of the program to use the generated API: the original programs create their goroutines and channels on the fly, whereas our adapted programs "pre-create" the goroutines and channels up front in a session initiation phase. In profiling, we find the actual computation code, which is the *same* in both versions, takes longer in the originals—one reason may be that the adapted versions run with better thread locality and fewer cache misses without such "interruptions" from goroutine spawning and channel creation.

6.2 Use Cases – Expressiveness and Applicability

We demonstrate the expressiveness and applicability of our framework by using our toolchain to specify and implement protocols for a range of role-parametric communication patterns, topologies and applications, listed in Table 15. The columns indicate the features of our extended Scribble used in the protocol. We cite the background and related works from where we draw the examples—in every case of parameterised session types literature, the parameterised aspect of the example was treated by either an *ad hoc* or *centralised* (non-distributed) method. The topologies in 4–8 are common in parallel algorithms. Due to space constraints, we explain the details of the examples in the Supplement,[4] § IV.1.

7 RELATED WORK

Parameterised MPST and implementations of session types. § 2.2 gave initial discussions of the closest related works on MPST for role-parametric protocols; we continue below.

Deniélou et al. [2012]; Yoshida et al. [2010] developed a role-parametric MPST using a dependent types approach. Unlike our work, the top-down *generic projection* in their theoretical-only work does not *infer* nor *decouple* role variants from the protocol; it simply encapsulates variant behaviours into a consolidated local type. To compensate, they combine with a *bottom-up* mechanism of taking endpoint decouplings from a pre-existing system of processes, and showing equivalence between the generic projection and target types; roughly speaking, however, for types that are "not syntactically close" (e.g., the generic projection of Ring and its role variants) the equivalence is often undecidable. In general, the programmers of individual endpoints in a modular development of some non-trivial multiparty application (e.g., not just binary RPCs) should commence development top-down from some notion of agreed protocol—otherwise the separate programmers cannot locally determine the (inherently stateful) I/O structure that their endpoint should implement.

Charalambides et al. [2016] extend MPST theory with parameterised versions of session type operators that represent repeat applications of the operator for some parameter value (possibly run-time instantiated). Unlike our work, their system does not support *role-parametric* protocols as their approach expressly requires prohibiting separate occurrences of a role with different indices; this rules out, e.g., role-parametric pipeline structures. Also, they did not implement their theory.

Regarding implementations and applications, Ng et al. [2015]; Ng and Yoshida [2015] use parameterised MPST [Deniélou et al. 2012] to generate an MPI backbone in C that encapsulates the *whole* protocol (i.e., every endpoint), and weaves (merges) it with user-supplied computation kernels. Their approach fundamentally produces complete, "centralised" programs, due to lacking notions of identifying and projecting role variants. By contrast, our toolchain generates typed APIs that allow the programmer to implement an individual endpoint more flexibly, i.e., not tied to a specific transport or messaging interface (MPI), nor a specific program structure.

López et al. [2015] develop a verification framework for MPI/C inspired by multiparty session types by translating parameterised protocol specifications to protocols in VCC [Cohen et al. 2009]. Their VCC verification is driven by program annotations, e.g., to match up individual control flow statements (e.g., `if-else`, `while`) to choices and loops in the specification, and pre/post conditions on recursive functions. Their approach is *purely global* (i.e., monolithic) from an MPST perspective: their specific aim is to verify a complete MPI program directly against a global protocol.

None of the remaining works in this paragraph support *parameterised* session types. Our API generation builds on the basic idea of Hu and Yoshida [2016] for Java, which our framework reformulates and extends for parameterised endpoints/families and nested FSMs in Go. Our API design leverages Go features that Java lacks (e.g., type switch, select); and is augmented in a range of ways, e.g., explicit error handling, nested struct types for peers/actions (which improves the IDE ergonomics of our APIs, while bypassing Go's lack of method overloading and reliance on singleton types), and promoting `End`-results to assist linearity. They did not evaluate run-time performance. Dynamic linearity checking is also employed in applications of session types in OCaml (Padovani [2017]) and Scala (Scalas et al. [2017]); our toolchain supports an alternative callback-based API generation that does not require dynamic checks. Gay et al. [2010] and Kouzapas et al. [2016] apply session types to object-oriented languages via *typestates* [Strom and Yemini 1986]. Unlike our API generation that targets programming in native Go, both are implemented as heavier-weight Java extensions with new syntax. (By contrast, the approach we use could possibly be described as *statetypes*.) As in our work and others above, these typestate approaches again rely on some form of cross-cutting linearity analysis.

Verification of message passing programs in Go. Our work aims to promote protocol compliance-by-construction in distributed programs through generation of types, to exploit lightweight error detection while programming and other support from IDEs and compile-time tools (e.g., "dot-driven" content assist and code auto-completion). Alternatively, the following are several recent works on *a posteriori* verification of message passing in existing Go programs. All of them employ *whole*-program techniques, and support only the built-in Go channel *primitives* (i.e., intra-process messaging); none of them, however, support channel-over-channel passing (§ 6.2).

Ng and Yoshida [2016] extract a graph-based protocol specification [Lange et al. 2015] from a Go program that is checked for deadlock-freedom; Stadtmüller et al. [2016] extract a regex-based protocol specification [Sulzmann and Thiemann 2016], checked for deadlock-freedom. Both approaches work only for programs restricted to *synchronous* Go channels; the former also requires all goroutines to be spawned before any communication among them occurs, and the latter has limited support for branching behaviours. Lange et al. [2017, 2018] statically infer channel communication patterns from a Go program as *behavioural types*, that are checked for liveness properties. The earlier work focuses on their analysis, a bounded symbolic method that does not scale well to large input models, and does not describe the inference procedure; it also does not take into account channel aliasing. The later work puts forward a concrete inference algorithm (for a restricted subset of Go) that considers channel aliasing. It checks the extracted types are restricted to finite control (not required in our work), which is required by a subsequent verification of the types by model checking; their model checker (mCRL2) also does not support channel passing, unlike our work (e.g., Pget). Their verification is best-effort only, due to the imprecision of the inference, and the verification times (and timeouts) preclude practical checking on the fly during programming.

The above works are the most related; we mention some further works in the Supplement[4] (§ V).

8 FUTURE WORK

We stated the conditions for concrete applications of our framework in § 2.1. We clarify further limitations relevant to our aims in this paper, and how they may be addressed in future work.

Dynamic participants. Our framework supports protocols where the (parameterised) participants are fixed on session initiation, as standard in MPST. We plan to integrate with explicitly (session-)typed *connection* actions [Hu and Yoshida 2017] for dynamic joining/leaving of parameterised participants during session execution; this would also eliminate some of the run-time connection checks at endpoint initiation (§ 5.3). To do so, we will extend our well-formedness based on the model checking approach of Hu and Yoshida [2017] for verifying MPST safety.

Failure handling. Our API generation is integrated with the explicit error handling paradigm of Go, where errors include node and networking failures. Our API design and safety guarantees currently consider the occurrence of such an error as a premature session termination (similar to linearity violations). We will investigate extending our framework to *fault-tolerant* protocols, e.g., for a session to continue between the remaining participants after one fails. We believe our formalism developed in this paper, that interprets our extensions in terms of a core base theory (§ 4), is well suited for such investigations: we may take one of the recent theoretical MPST works on link failures [Adameit et al. 2017] or crash failures [Viering et al. 2018] as a base theory.

Programming styles. This paper focuses on an API style that is close to channel-based programming using standard libraries and Go channels; our aim is to offer MPST-based programming through a familiar interface to Go users, and to facilitate the reimplementation of existing Go programs for our evaluation. The presented APIs promote a popular call-chaining programming style (cf. fluent APIs) that permits some flexibility between more "imperative" or more "functional" styles within the context of Go. We briefly illustrated our alternative callback-based API generation, that inherently precludes run-time linearity checks, but requires programming in an event-driven style—also a widely used style in practice. We plan to add further API generation styles, such as a "monadic" or CPS-based style that relies less on side effects for input methods (cf. Fig. 5, line 17). We note the *necessary* language features to implement (a basic version of) our approach are relatively modest support for static typing of data and functions/methods. We have leveraged additional Go specific features to produce better user APIs (e.g., type-switch and goroutines), but they are inessential. We believe our approach may be readily ported to other languages, given that we have demonstrated an implementation for Go whose type system is (by design) relatively bare.

Acknowledgements. We thank the anonymous reviewers and Simon Castellan for their feedback. This work is partially supported by EU COST Action IC1402 ARVI, EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1.

REFERENCES

Manuel Adameit, Kirstin Peters, and Uwe Nestmann. 2017. Session Types for Link Failures. In *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10321. Springer, 1–16. https://doi.org/10.1007/978-3-319-60225-7_1

Foat Akhmadeev. 2016. Web Crawling With Akka. http://foat.me/articles/crawling-with-akka/.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo DeniÃĺlou, 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. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

Daniel Austin, Abbie Barbir, Ed Peters, and Steve Ross-Talbot. 2004. Web Services Choreography Requirements. https://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/#UC-002.

Andi Bejleri, Raymond Hu, and Nobuko Yoshida. 2009. Session-Based Programming for Parallel Algorithms: Expressiveness and Performance. In *Proceedings Second International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2009, York, UK, 22nd March 2009. (EPTCS)*, Vol. 17. 17–29. https://doi.org/10.4204/EPTCS.17.2

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

Stefan Brunthaler. 2010. Inline Caching Meets Quickening. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 429–451. https://doi.org/10.1007/978-3-642-14107-2_21

Minas Charalambides, Peter Dinges, and Gul A. Agha. 2016. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.* 115-116 (2016), 100–126. https://doi.org/10.1016/j.scico.2015.10.006

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS)*, Vol. 9104. Springer, 146–178.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science)*, Helmut Seidl (Ed.), Vol. 7211. Springer, 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.), Vol. 7966. Springer, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

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

Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular Session Types for Distributed Object-oriented Programming. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 299–312. https://doi.org/10.1145/1706299.1706335

Issac Gouy. 2017. Computer Language Benchmark Game. http://benchmarksgame.alioth.debian.org.

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

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Perdita Stevens and Andrzej Wasowski (Eds.), Vol. 9633. Springer, 401–418. https://doi.org/10.1007/

978-3-662-49665-7_24
Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Marieke Huisman and Julia Rubin (Eds.), Vol. 10202. Springer, 116–133. https://doi.org/10.1007/978-3-662-54494-5_7
Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052
Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *PPDP*. 146–159. https://doi.org/10.1145/2967973.2968595
Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off go: liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 748–761. http://dl.acm.org/citation.cfm?id=3009847
Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*. ACM. To appear.
Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 221–232. https://doi.org/10.1145/2676726.2676964
Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 280–298. https://doi.org/10.1145/2814270.2814302
Rumyana Neykova and Nobuko Yoshida. 2017. Let It Recover: Multiparty Protocol-Induced Recovery. In *26th International Conference on Compiler Construction*. ACM, 98–108.
Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. 2015. Protocols by Default - Safe MPI Code Generation Based on Session Types. In *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Björn Franke (Ed.), Vol. 9031. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11
Nicholas Ng and Nobuko Yoshida. 2015. Pabble: parameterised Scribble. *Service Oriented Computing and Applications* 9, 3-4 (2015), 269–284. https://doi.org/10.1007/s11761-014-0172-8
Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 174–184. https://doi.org/10.1145/2892208.2892232
Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289
Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 655–670. https://doi.org/10.1109/SP.2014.48
Victor Samofalov, V. Krukov, B. Kuhn, S. Zheltov, Alexander V. Konovalov, and J. DeSouza. 2005. Automated Correctness Analysis of MPI Programs with Intel(r) Message Checker. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain (John von Neumann Institute for Computing Series)*, Vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 901–908.
Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. 2009. Phaser accumulators: A new reduction construct for dynamic parallelism. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 1–12. https://doi.org/10.1109/IPDPS.2009.5161071
Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization coaching: optimizers learn to communicate with programmers. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 163–178. https://doi.org/10.1145/2384616.2384629

Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. 2016. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 116–136. https://doi.org/10.1007/978-3-319-47958-3_7

R E Strom and S Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.* 12, 1 (Jan. 1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Martin Sulzmann and Peter Thiemann. 2016. Forkable Regular Expressions. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings (Lecture Notes in Computer Science)*, Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe (Eds.), Vol. 9618. Springer, 194–206. https://doi.org/10.1007/978-3-319-30000-9_15

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 447–458. https://doi.org/10.1145/1926385.1926436

Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. 2018. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 799–826. https://doi.org/10.1007/978-3-319-89884-1_28

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 377–388. https://doi.org/10.1145/1706299.1706343

Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. 2010. Parameterised Multiparty Session Types. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6014. Springer, 128–145. https://doi.org/10.1007/978-3-642-12032-9_10