Dynamically Updatable Multiparty Session Protocols

Generating Concurrent Go Code from Unbounded Protocols

David Castro-Perez
University of Kent, UK

Nobuko Yoshida
University of Oxford, UK

Abstract

Multiparty Session Types (MPST) are a typing disciplines that guarantee the absence of deadlocks and communication errors in concurrent and distributed systems. However, existing MPST frameworks do not support protocols with *dynamic unbounded participants*, and cannot express many common programming patterns that require the introduction of new participants into a protocol. This poses a barrier for the adoption of MPST in languages that favour the creation of new participants (processes, lightweight threads, etc) that communicate via message passing, such as Go or Erlang.

This paper proposes *Dynamically Updatable Multiparty Session Protocols*, a new MPST theory (DMst) that supports protocols with an *unbounded* number of fresh participants, whose communication topologies are *dynamically updatable*. We prove that DMst guarantees deadlock-freedom and liveness. We implement a toolchain, GoScr (Go-Scribble), which generates Go implementations from DMst, ensuring **by construction**, that the different participants will only perform I/O actions that comply with a given protocol specification. We evaluate our toolchain by (1) implementing representative parallel and concurrent algorithms from existing benchmarks, textbooks and literature; (2) showing that GoScr does not introduce significant overheads compared to a naive implementation, for computationally expensive benchmarks; and (3) building three realistic protocols (dynamic task delegation, recursive Domain Name System, and a parallel Min-Max strategy) in GoScr that could not be represented with previous theories of session types.

2012 ACM Subject Classification Theory of computation → Program specifications; Computing methodologies → Concurrent programming languages

Keywords and phrases Multiparty Session Types, Correctness-by-construction, Concurrency, Golang

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.10

Acknowledgements We deeply thank Benito Echarren Serrano for his initial collaboration on a preliminary version of this work. This work is supported by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

1 Introduction

Multiparty Session Types. Multiparty Session Types (MPST) are typing disciplines that can guarantee the absence of deadlocks and communication errors in concurrent and distributed systems [21, 22]. MPST allow the specification of *global communication protocols* (**global types**) among a number of participants. The **projection** operation extracts the *local communication protocols* (**local types**), from the point of view of each participant in the system. Projection only succeeds when the protocol is absent of deadlocks and communication errors. These local types can then be used to typecheck processes [21], generate **correct by construction** code [25, 3], or monitor to detect communication errors at runtime [8].

However, MPST have a severe limitation: they cannot model protocols in which *new participants join the system*. Many important protocols rely on this. For example, Chord [54] is a popular protocol for distributed hash tables where participants join a ring, and relies on a stabilisation protocol to guarantee that each participant keeps up-to-date channels to their successors and predecessors. To model such scenarios using MPST, it would be necessary to *interleave* different sessions. But arbitrary session interleavings can lead to deadlocks, so it must be restricted [2, 5]. This not only rules out the use of MPST for many realistic scenarios, but also limits the applicability of MPST for languages that favour process creation and message passing, such as **Go**, which is the main motivation of our work.

Dynamic (Unbounded) Participants in Go. Go is a concurrent programming language designed in 2009 by Google, and it is increasingly popular among professional developers. According to a 2020 Stack Overflow survey, Go is used by 9.4% of developers, and it is the "third most wanted language" [52]. Go was also the 4th most active language in GitHub in 2020 [16], and it has been adopted in many large software systems such as Kubernetes [32], gRPC [18] and Docker [13]. Its main features are explicit communication primitives, namely *channels* and *goroutines* (lightweight threads), whose design comes from concurrent process calculi [20, 40, 41]. Unfortunately, a recent empirical study reveals that over 50% of Go concurrent bugs are caused by communication [56, 39, 61] (i.e., more than shared memory bugs). While Go includes a global *runtime* deadlock detector, it is neither adequate to verify applications with complex communication structures, nor can it detect deadlocks involving only a strict subset of a program's goroutines (partial deadlocks) [37].

Figure 1 illustrates Go's core concurrency constructs. It shows a server (Master) that processes client requests (Line 4), and sends responses back to the Client (Line 20). The Master breaks down the request into different subtasks and delegates them to different Worker goroutines (Lines 7–10). The Master then aggregates the Worker results (Lines 11–19). If the Master receives an error message, it will forward it to the Client and stop processing any new messages (Lines 16–19). This program uses a common Go computation pattern[1], the master-worker pattern, and the number of workers depends on a **runtime value**.

Unfortunately, there is a bug in the implementation in Figure 1. The implementation uses synchronous channels. Since the Master goroutine stops processing Worker responses after receiving the first error message, all other goroutines which have not sent their result or error messages will be **deadlocked**, as they will be stuck waiting for the Master to process their message. One might think that this error could be fixed by replacing the synchronous channels in the implementation with asynchronous (buffered) channels. Unfortunately, this approach leaves *orphan messages* which could introduce other concurrency bugs, e.g. the Master may need to clean up resources after receiving a response from the Workers.

This example demonstrates how even in simple programs, message passing can introduce concurrency bugs and channel leakage, violating **deadlock-freedom** and **liveness**. While, in simple programs, these concurrency bugs can be fixed with relative ease, identifying and fixing them is usually done during testing phase, which becomes increasingly harder as the complexity of the program and the number of goroutines increases. Unfortunately, standard MPST cannot model protocols such as Figure 1, since the number of participants is not fixed at the start, and depends on a run-time value.

Adding dynamic participants to MPST. This paper introduces *Dynamically Updatable Multiparty Session Types* (DMst), a new theory of MPST whose **novel feature** is to model protocols in which participants can join an already existing session (*dynamic participants*). DMst can guarantee *deadlock-freedom* and *liveness (partial-deadlock-freedom)* **by construction** in such protocols. We implement DMst as a tool, GoScr, which generates

[1] E.g. `https://github.com/tmrts/go-patterns/blob/master/messaging/fan_out.md`

```
func Worker(n int, resp chan int, err chan error) { ... } // Worker returns either result or error
func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
  for {
    ubound := <-reqCh                                   // Receive request from Client
    workerChs := make([]chan int, ubound)               // Array to store worker result channels
    errCh := make(chan error)
    for i := 0; i < ubound; i++ {                       // n_workers depends on runtime value
      workerChs[i] = make(chan int)                     // Create worker channel
      go Worker(i+1, workerChs[i], errCh)
    }
    var res []int
    for i := 0; i < ubound; i++ {                       // Aggregate worker results
      select {
      case sql := <-workerChs[i]:                       // Aggregate successful result
        res = append(res, sql)
      case err := <-errCh:                              // Some worker failed
        cErrCh <- err                                   // Propagate error and
        return                                          // stop processing any further messages
      }}
    respCh <- res}}                                     // Send final result to client
```

Figure 1 Dynamic task delegation implementation in Go (unsafe)

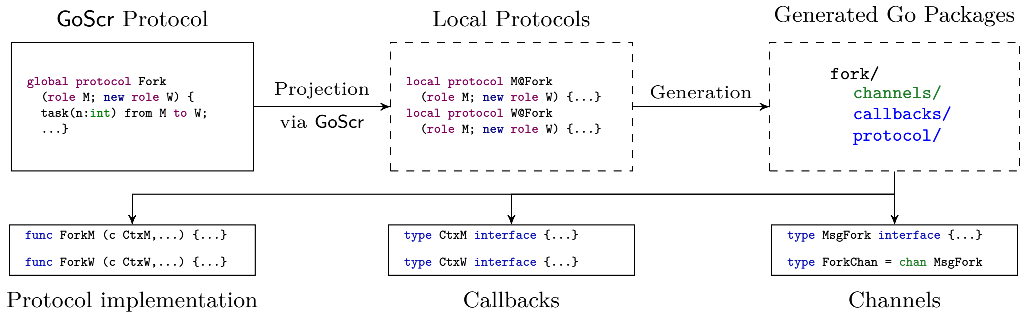
correct by construction Go code, and we evaluate it on a number of representative algorithms in Go, including a safe version of Figure 1 (see § 5.2(a)). While our target language is Go, DMst is not Go specific and a part of GoScr (GoScr protocols, projection and local protocols in Figure 2) is reusable for any language, as long as it supports (1) the creation of new participants (threads or processes) and (2) communication between participants.

Contributions. DMst overcomes several bottlenecks of existing theories on session types **(A)**; and the two main lines of work **(B,C)** for static deadlock detection in Go:

(A) Dynamic Participants and Session Types. There are two main existing theoretical lines of work related to dynamic MPST. *Dynamic Multirole Session Types* (MRST) [9] enable a set of participants which belong to the same group (i.e. role) to join a multiparty session type. The roles are *fixed at the start*, and can only join at specific points in the protocol, e.g at the beginning of each protocol iteration. *Nested MPST* [7] model protocols with unbounded new participants. Neither MRST nor Nested MPST can represent DMst protocols **(A-1)** where participants join dynamically to recursive protocols, except at fixed points and with fixed roles (see Example 6). In addition, **(A-2)** our theory provides stronger guarantees than [7], while their global types are more complex, as they must be checked by a complex typing system. Hence a safe version of Figure 1 cannot be represented by [7, 9]. Both of [9, 7] are *only theoretical*, and lack any implementation or practical results. DMst's global types are not only more expressive than those in [9, 7], but also simpler, thus DMst is more suitable for real language implementations. Other lines of work add session types to calculi that allow dynamic participants, or extend MPST to specify where can participants join in a protocol, e.g. [19, 57, 58, 30]. While these lines of work can add or replace participants to a system, these participants must act according to known, fixed roles. Therefore, these lines of work do not allow the specification of cyclic recursive topologies that change dynamically with the introduction of new participants.

(B) Inference Approach. This approach verifies safety and liveness properties of Go programs, by using model-checking on their *inferred* concurrent behavioural types [47, 36, 37, 14]. The major limitations of this approach are: **(B-1)** there is a gap between properties of types and programs, i.e., there are cases where types satisfy liveness but programs do not, leading to *unsound* verification, and **(B-2)** it cannot verify infinitely spawning goroutines because either the theory is limited to bounded approximation [36] or a decidable set of types are limited to finite-control (i.e. no parallel processes inside loops) [37, 14].

(C) Go Code Generation. Another approach is the generation of Go code from



```
global protocol Fork
  (role M; new role W) {
  task(n:int) from M to W;
  ...}
```

```
local protocol M@Fork
  (role M; new role W) {...}
local protocol W@Fork
  (role M; new role W) {...}
```

```
fork/
  channels/
  callbacks/
  protocol/
```

```
func ForkM (c CtxM,...) {...}

func ForkW (c CtxW,...) {...}
```

```
type CtxM interface {...}

type CtxW interface {...}
```

```
type MsgFork interface {...}

type ForkChan = chan MsgFork
```

Figure 2 Overview of GoScr toolchain

parameterised multiparty session protocols [3]. However, the major limitation of [3] is that participants in a protocol still need to be **fixed** at the start of a session, so it cannot express and generate code for typical Go-style programs with goroutines – e.g. a safe version of Figure 1. There is a subtle, but important distinction between dynamic participants and parameterised roles: parameterised roles cannot depend on a run-time value that is exchanged in a message *that is part of the protocol*, because in parameterised MPST approaches, all of the participants must join the session at session initialisation, and are therefore fixed.

Our challenges are to overcome all these limitations with a scalable (implementable) MPST theory. In summary, this work solves bottlenecks of the existing MPST work by proposing a new theory, DMst, that allows the dynamic generation of an unbounded number of participants in recursive protocols, overcoming expressiveness issues in [9, 7] **(A)** and **(C)**; unsoundness **(B-1)**, but is not limited to a bounded analysis nor finite-control **(B-2)**.

Outline. **§ 2** presents an overview of the GoScr toolchain; **§ 3** presents DMst, multiparty session types extended with the ability to add unbounded participants dynamically during a protocol execution, and proves its deadlock-freedom (Theorem 23), orphan message freedom, and liveness (Theorem 29); **§ 4** describes the code generation process of GoScr, and how to use it to implement DMst protocols; **§ 5** first measures the runtime overhead of the GoScr backend, then demonstrate the expressiveness of DMst, comparing the expressiveness of GoScr to **(A)** [47, 36, 37, 14] and **(B)** [3] with a number of case studies. We also implement three use cases – dynamic task delegation, a recursive Domain Name System, a noughts and crosses game with Min-Max strategy – to demonstrate the applicability of GoScr; **§ 6** gives related work, and **§ 7** concludes with future work.

2 Overview of GoScr

GoScr follows a typical Multiparty Session Types workflow (see diagram on the right). In this workflow, the starting point is the definition of a *global protocol* (*global type* in MPST), which describes a structured sequence of interactions between a number of participants. From this global type, we extract automatically a number of *local protocols* (*local types*) that describe the interactions (i.e. send or receive actions) from the point of view of

$$\begin{tikzcd}
\text{projection} & & & G \arrow[dll] \arrow[dl] \arrow[d, dashed] \arrow[dr] & \\
\text{code generation} & L_1 \arrow[d] & L_2 \arrow[d] & \cdots \arrow[d, dashed] & L_n \arrow[d] \\
 & P_1 & P_2 & \cdots & P_n
\end{tikzcd}$$

every participant in the protocol. This is done using the *projection* operation. If some participant is not projectable, then we raise an error, since the protocol is not well-formed and can lead to deadlocks or other communication errors. If the programmer provides a set of processes that behaves as prescribed by each of the local types, then the whole system

```
1  global protocol UPipe(role M;new role W){
2    rec X {
3      choice at M {
4        (Put:int) from M to W;
5         continue X with W calls UPipe(W);
6      } or {
7        (Quit:int) from M to W; }}}
8  local protocol M@UPipe(role M;new role W){
9    rec X {
10     choice at M {
11       (Put:int) to W;
12       continue X;
13     } or {
14       (Quit:int) to W; }}}
```

```
local protocol W@UPipe(role M;new role W){    1
  choice at M {                                 2
    (Put:int) from M;                           3
    invite UPipe(self; new W2);                 4
    rec X {                                     5
      choice at M {                             6
        (Put:int) from M;                       7
        (Put:int) to W2;                        8
        continue X;                             9
      } or {                                   10
        (Quit:int) from M;                     11
        (Quit:int) to W2;                      12
      }}                                       13
  } or { (Quit:int) from M; }}                 14
```

Figure 3 Global and Local protocols for a dynamic recursive pipeline.

is safe. We take a *code generation* approach, where we generate process code from their respective local types, providing safety guarantees *by construction*.

Adding Participants Dynamically. GoScr is a code generation tool which extends nuScr [48] with the theory of DMst, targeting the Go language. nuScr is a new implementation of Scribble [51], aimed at experimenting with extensions to core MPST. Figure 2 presents an overview of GoScr. We distinguish toolchain internals (dashed boxes) from tool inputs ($\boxed{\text{solid boxes}}$). Development starts by specifying a global protocol in GoScr [51, 23], a programmer-friendly protocol description language based on MPST [22, 44]. GoScr validates the well-formedness of the protocol, and produces a local type for each participant via *projection*. GoScr generates protocol implementations from these sets of local types. We provide an overview of the GoScr workflow using the dynamic recursive pipeline of Figure 3. Intuitively, this pipeline introduces a new participant after each iteration.

Global Protocol Specification. The first key novel feature of GoScr is the ability to *define* and *call* protocols (e.g. Line 1 in Figure 3) that may bring new participants to the protocol *dynamically*, specified by the `new` keyword in the signature. These calls can be recursive, allowing for an *unbounded* number of participants. Lines 1–14 declare `UPipe`, which requires only participant `M`, and introduces a new participant `W` dynamically. Protocol calls create any necessary new participants, as well as any necessary channels, before performing the interactions described by the called protocol. The second key novel feature of GoScr is the ability to *modify* a recursive protocol by *combining* (i.e. *interleaving*) its interactions with those of a protocol call. In our syntax, this is specified by *annotating* recursion variables with protocol calls. We call this ***updatable recursion***, and an example of this can be found in Line 5. The meaning of such calls is as follows. Suppose that processes $\mathbf{r}_0$ and $\mathbf{r}_1$ are behaving as `M` and `W` (resp.) in `UPipe`. Just before the protocol jumps back to Line 2, process $\mathbf{r}_1$ calls protocol `UPipe`($\mathbf{r}_1$). This means that $\mathbf{r}_1$ will create a new participant $\mathbf{r}_2$, and $\mathbf{r}_1$ will *delegate* to $\mathbf{r}_2$ a session to act as `W` in `UPipe`, with $\mathbf{r}_1$ acting as `M`. But at this point, $\mathbf{r}_1$ should act as both `M` and `W`. To address this, $\mathbf{r}_1$ will combine its interactions acting as `M` and `W`. The fact that $\mathbf{r}_1$ needs to change its behaviour to act as *two* distinct roles in `UPipe` will be reflected in its ***local protocol specification*** (participant `W` in Figure 3).

Local Protocol Specification. GoScr extracts local protocol specifications from global protocols using an operation called *projection*. Local protocols describe the structured sequence of interactions, from the point of view of a single participant. Figure 3 lists local protocols for `M` and `W`. Consider the point of view of the new participant `W` in protocol `UPipe` from Figure 3. `W` first receives an integer, either with label `Put` or `Quit` from `M`. If `W`

```
1  type Put int
2  type Quit int
3  type Ctx_UPipe_W interface {
4    Recv_M_Put(v_2 Put)
5    Init_W_UPipe() Ctx_UPipe_W
6    ...
7    Recv_M_Quit(v_2 Quit)
8    Quit() }
9  func UPipeW(ctx Ctx_UPipe_W,
10   wg *sync.WaitGroup, chMW chan MsgUPipe){
11   defer wg.Done()
12   x_1 := <- chMW
13
14   switch v_2 := x_1.(type) {
15   case Put:
16     ctx.Recv_M_Put(v_2)
17     ch_W_W_1 := make(chan MsgUPipe, 1)
18     ctx_1 := ctx.Init_W_UPipe_Ctx()
19     wg.Add(1)
20     go UPipeW(ctx_1,wg, ch_W_W_1)
21 MuX:
22     for {
23       ... }
24   case Quit:
25     ctx.Recv_M_Quit(v_2)
26     ctx.End() }}
```

Figure 4 Implementation and context of role `W` in `UPipe` in Figure 3.

receives `Quit`, then the protocol finishes. Otherwise, `W` performs a protocol call, bringing in a new participant `W2` to act as `W` in `UPipe`. In the subsequent interactions, from Line 5, `W` acts as both `M` (with respect to `W2`) and `W` (with respect to `M`). These lines (5 – 13) appear as a result of projecting Line 5 onto `W`. Notice that, if we have two participants, one acting as `M` and another one acting as `W`, this will generate a pipeline with an unbounded number of stages, until the first participant acting as `M` sends `Quit`. These kinds of protocols could not be represented in previous MPST theories and frameworks.

Program Logic. From local protocol specifications, GoScr generates the implementation of each role as a self-contained function. GoScr interleaves communication actions and the program logic. Communication actions in Go are a direct translation of those in local protocols: a send is a regular Go send, a receive is a regular Go receive, a choice is a type switch on a label, etc. Programmer inputs at this stage are, therefore, protocol specifications and program logic. We follow a *callback* approach similar to [42, 62] that guarantees correctness of communication *by construction*, unlike other approaches that required runtime *linearity* checks [3]. We discuss this approach in detail in §4. Figure 4 presents the code that GoScr generates for `W` in `UPipe`. The generated implementation requires that the programmer implements the **context** interface Ctx_UPipe_W (Lines 3 – 8, Figure 4). This interface defines all the necessary callbacks to implement the program logic. Programmers can use any type definition to store a local state for each participant in the protocol, e.g.

```
type CtxW int // This type implements Ctx_UPipe_W, and stores the accumulated sum
func (c *CtxW) Recv_M_Put(v upipe.Put)  { // upipe.Put is also an 'int'
  *c += CtxW(v) }
...
```

By using CtxW for implementing Ctx_UPipe_W, workers will store the sum of all the numbers that they receive, and forward their accumulated sum to the next participant. The generated code for `W` will signal when it has terminated (Line 11), and starts by receiving from `M` (Line 12). Depending on whether `W` receives Put or Quit, `W` continues with the corresponding branch (Line 14). If `M` sends Put, then `W` creates a new participant that also acts as `W` (with respect to the previous `W`). To create this participant, first a channel is created (Line 17), then a new context is created (Line 18), the participant count is increased to guarantee that execution does not end before all participants have ended (Line 19), and finally a new goroutine is created (Line 20). Otherwise, if `M` sends Quit, the callback for ending is called, a last callback to perform any necessary cleanup is called, and the participant ends (Lines 25 and 26).

As we show in Figure 2, from a global protocol specification GoScr produces an implementation of all of its participants. To run this generated implementation, programmers must provide the necessary types to represent protocol contexts and their required callbacks. Our code generation scheme statically ensures that implementations never lead to the errors described in § 1, i.e. there will be no **deadlocks** and **orphan messages**.

3 Dynamically Updatable Unbounded Multiparty Session Protocols

This section introduces the theory of *Dynamically Updatable Multiparty Session Types* (DMst) with examples, and proves that DMst satisfies deadlock-freedom and liveness. DMst is the formalism that underlies GoScr. To illustrate our theory, consider the dynamic pipeline of Figure 3. In this protocol, new participants are introduced into the protocol after each iteration. In DMst, we write this dynamic pipeline as follows:

$$\mathit{Pipe} = \lambda\langle \mathbf{p}; \nu\mathbf{q}\rangle.\mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow \mathit{Pipe}\langle\mathbf{q}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})$$

This protocol definition requires two participants $\mathbf{p}$ and $\mathbf{q}$. Participant $\mathbf{q}$ is annotated with ν to specify that it is introduced *dynamically* (a *dynamic participant*). Participant $\mathbf{p}$ is called a *parameter participant*. The body of the protocol specifies that it is a recursive protocol ($\mu\mathbf{t}.\ldots$), with recursion variable $\mathbf{t}$, where $\mathbf{p}$ sends to $\mathbf{q}$ either put or quit. This is a choice ($+$), where each branch starts with $\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}]$ and $\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}$ respectively. If $\mathbf{p}$ sends put, then both participants enter a new iteration, but $\mathbf{q}$ *extends* the protocol by performing call to Pipe ($\mathbf{q} \hookrightarrow \mathit{Pipe}\langle\mathbf{q}\rangle$) before entering the new iteration. Note that although the signature mentions two participants $\mathbf{p}$ and $\mathbf{q}$, the call in the global type only needs to list the parameter participants. This protocol call effectively brings in a new participant to the protocol (e.g. $\mathbf{r}$), creates and distributes the necessary additional channels, and extends the interactions of the protocol with those of $\mathit{Pipe}(\mathbf{q}; \mathbf{r})$.

Introducing new interactions into an existing protocol requires to *interleave* them with the actions of this existing protocol. For example, the interactions of $\mathit{Pipe}(\mathbf{q}; \mathbf{r})$ need to be interleaved with the remaining interactions of $\mathit{Pipe}(\mathbf{p}; \mathbf{q})$. Our protocol specification allows two forms of interleavings: (a) sequencing all the interactions of a protocol call with the remaining interactions; and (b) alternating the actions of each iteration of two recursive protocols. We introduce a protocol construct $\blacklozenge$ to specify the latter.

3.1 Global Types of DMst

The syntax of DMst global types (given in Definition 1) is an extension of the simplest version of MPST [60]. The novel added features are highlighted.

▶ **Definition 1** (DMst Global Types).

$$\gamma ::= \mathbf{p} \to \mathbf{q}{:}m[U] \mid \mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle \qquad G ::= \mathsf{end} \mid \gamma.G \mid \textstyle\sum_{i \in I} G_i \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid G \blacklozenge \vec{\gamma}$$

Prefixes (γ, γ', ...) represent individual interactions between ***participants***, also called ***roles***[2], ($\mathbf{p}, \mathbf{q}, \mathbf{r}, \ldots$). There are two prefixes: messages and protocol calls. A ***message*** between $\mathbf{p}$ and $\mathbf{q}$ with ***label*** m and ***payload type*** U (e.g. int, bool, ...) is written $\mathbf{p} \to \mathbf{q}{:}m[U]$, or $\mathbf{p} \to \mathbf{q}{:}m$ whenever the payload is not relevant, e.g. when U is unit. We write $\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle$ to denote a ***call*** to protocol x by $\mathbf{p}$, with participants $\vec{\mathbf{q}}$ ($= \mathbf{q_1} \ldots \mathbf{q_n}$) (see ***protocol definitions*** below). A protocol call prefix will introduce the new interactions described by x.

Global types ($G, G', \ldots$) denote global protocols among participants. The syntax of global types is mostly standard: end is ***termination*** and it is often omitted. $\mathbf{t}$ denotes a ***recursive variable***. ***Choice*** $\sum_{i \in I} G_i$ chooses any G_i, depending on the first action of each G_i (see Definition 3). ***Recursive protocol*** $\mu\mathbf{t}.G$ behaves as G, binding recursive variable $\mathbf{t}$ to $\mu\mathbf{t}.G$. ***Sequencing*** $\gamma.G$ denotes the execution of a prefix γ, and a continuation G. The new construct $G \blacklozenge \vec{\gamma}$ denotes an ***updatable protocol***, where G is extended with the interactions

[2] A participant plays a *role* in the protocol, and this role is determined by the structured sequence of interactions that are allowed by the global type.

and participants introduced by $\vec{\gamma}$ (if any). When G is a recursive variable $\mathbf{t}$ ($\mathbf{t} \blacklozenge \vec{\gamma}$), we often call these *updatable recursion*, or *updatable recursion variable.* We use updatable protocols to represent recursive protocols where subsequent iterations are extended with new message exchanges and/or participants. We will show in Example 6 how to use updatable recursion to represent the dynamic recursive pipeline of Figure 3.

Choice well-formedness. Standard MPST syntax only allows choices where a participant $\mathbf{p}$ sends to another participant $\mathbf{q}$ a distinct label in each branch. This means that $\mathbf{p}$ and $\mathbf{q}$ can use the label to distinguish each branch of the choice [21, 60]. DMst's syntax is more flexible, since branches can also be distinguished by distinct protocol calls. However, we still require that a single participant either sends a distinct label, or performs a distinct protocol call as the first interaction of each branch. We say that the choices that satisfy this condition are *directed.* Checking that choices are directed is necessary for well-formedness, but it is not sufficient. Protocol well-formedness is defined in a standard way later in Definition 15. To refer to the interaction that occurs in a branch, we use the *extended labels.*

▶ **Definition 2** (Extended Labels). *We define* extended labels, $\ell ::= m \mid i@x(\vec{\mathbf{p}};\vec{\mathbf{q}})$, *where* $i@x(\vec{\mathbf{p}};\vec{\mathbf{q}})$ *identifies a protocol call as the* i*-th participant of* x *with participants* $\vec{\mathbf{p}};\vec{\mathbf{q}}$. *We use participant index instead of name, since* x *may give different names to* $\vec{\mathbf{p}}$ *and* $\vec{\mathbf{q}}$.

▶ **Definition 3** (Directed Choices). Then, we define $\mathtt{dc}$ (directed choice):

$$\mathtt{dc}(\mathbf{p}, \{\ell_i\}_{i\in I}, \textstyle\sum_{i\in I} \gamma_i.G_i) = (\forall i \in I.\mathtt{inter}(\mathbf{p}, \ell_i, \gamma_i)) \text{ with all } \ell_i \neq \ell_j \text{ for } i \neq j$$

The predicate $\mathtt{inter}(\mathbf{p}, \ell_i, \gamma_i)$ states that γ_i is an interaction initiated by $\mathbf{p}$ with extended label ℓ_i: $\mathtt{inter}(\mathbf{p}, i@x(\vec{\mathbf{p}};\vec{\mathbf{q}}), \mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle)$, if $i \leq \mathsf{size}(\vec{\mathbf{p}}\vec{\mathbf{q}})$, and $\mathtt{inter}(\mathbf{p}, m, \mathbf{p} \to \mathbf{q}{:}m[U])$.

Protocol definitions ($x = \lambda\langle\vec{\mathbf{q}}; \nu\vec{\mathbf{r}}\rangle.G$) associate a protocol name x with a global type G, given a sequence of parameter participants $\vec{\mathbf{q}}$, and a sequence of *new* participants $\vec{\mathbf{r}}$ (where "ν" means "new" [41]) that join the protocol ***dynamically*** (we call these *dynamic participants*). Any participant occurring in G must be bound by $\vec{\mathbf{q}}$ or $\vec{\mathbf{r}}$. Protocol call prefixes ($x\langle\vec{\mathbf{q}}\rangle$) only specify the parameter participants, not the dynamic ones. To refer to the global type of a definition, we write $x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$, with parameter participants $\vec{\mathbf{q}}$, and dynamic participants $\vec{\mathbf{r}}$.

▶ **Example 4** (Fibonacci). The following protocol represents the interactions of an unbounded series of participants, that together compute the Fibonacci sequence:

$$\mathit{Fib} = \lambda\langle\mathbf{s}, \mathbf{f_1}, \mathbf{f_2}; \nu\mathbf{f_3}\rangle.\mathbf{f_1} \to \mathbf{f_3}{:}\mathsf{F}[\mathsf{int}].\mathbf{f_2} \to \mathbf{f_3}{:}\mathsf{F}[\mathsf{int}].\mathbf{f_3} \to \mathbf{s}{:}\mathsf{NF}[\mathsf{int}].\mathbf{f_3} \hookrightarrow \mathit{Fib}\langle\mathbf{s}, \mathbf{f_2}, \mathbf{f_3}\rangle.\mathsf{end}$$

Fib defines a protocol that recursively creates new participants ($\mathbf{f_3}$ in the global type) to compute the next element of the Fibonacci sequence after receiving the results from the previous two participants ($\mathbf{f_1}$ and $\mathbf{f_2}$). Participant $\mathbf{s}$ receives all the results. Intuitively, the implementation of $\mathbf{f_3}$ starts by receiving from $\mathbf{f_1}$ and $\mathbf{f_2}$, sends the new Fibonacci number to $\mathbf{s}$, and then creates a new participant and continues with $\mathbf{f_2}$ acting as $\mathbf{f_1}$, and $\mathbf{f_3}$ as $\mathbf{f_2}$. The code generated by a similar protocol is shown later in Figure 5.

Protocol calls can also be used to represent recursive protocols that are augmented dynamically with new interactions and/or participants. To represent such protocols we use updatable recursion variables. Intuitively, subsequent iterations of a recursive protocol $\mu\mathbf{t}.G$ that contains an updatable recursion variable $\mathbf{t}\blacklozenge\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle$ will proceed as $\mu\mathbf{t}.G$ *combined* with the global type defined by x. Global types are combined by interleaving their interactions.

▶ **Definition 5** (Combining Recursive Global Types). *Let* cont *be a function that computes the set of final continuations, i.e. recursion variables or* end*, after executing all possible prefixes:* $\mathsf{cont}(\gamma.\, G) = \mathsf{cont}(G)$, $\mathsf{cont}(\mu\mathbf{t}.G) = \mathsf{cont}(G) \setminus \{\mathbf{t}\}$, $\mathsf{cont}(\sum_{i\in I} G_i) = \cup_{i\in I}\mathsf{cont}(G_i)$, $\mathsf{cont}(\mathbf{t}) = \{\mathbf{t}\}$, $\mathsf{cont}(G \blacklozenge \gamma) = \mathsf{cont}(G)$, $\mathsf{cont}(\mathsf{end}) = \{\mathsf{end}\}$. *We define*

$$(\mu\mathbf{t}.\textstyle\sum_{i\in I} G_i') \between (\mu\mathbf{t}.\sum_{i\in I} G_i) = \mu\mathbf{t}.\sum_{i\in I} (G_i' \between_{\mathbf{t}} G_i)$$

where $G' \between_{\mathbf{t}} G = [G/\mathbf{t}]G'$ *if* $\mathsf{cont}(G') = \mathsf{cont}(G) = \{\mathbf{t}\}$, $G' \between_{\mathbf{t}} G = [G/\mathsf{end}]G'$ *if* $\mathsf{cont}(G') = \mathsf{cont}(G) = \{\mathsf{end}\}$, *and is undefined otherwise.*

The composition operator takes two recursive protocols with the same branching structure, and combines each of the branches using $G' \between_{\mathbf{t}} G$. This operator simply appends the interactions of G after the interactions of G' by substituting either end or $\mathbf{t}$ by G. Both G and G' must finish with the same last continuation, either $\mathbf{t}$ or end. For example:

$$((\gamma_1.\ \mathsf{end}) + (\gamma_2.\ \mathbf{t})) \between_{\mathbf{t}} ((\gamma_3.\ \mathsf{end}) + (\gamma_4.\ \mathbf{t})) = (\gamma_1.\ \gamma_3.\ \mathsf{end}) + (\gamma_2.\ \gamma_4.\ \mathbf{t}),$$

but the following case is undefined: $((\gamma_1.\ \mathsf{end}) + (\gamma_2.\ \mathbf{t})) \between_{\mathbf{t}} ((\gamma_3.\ \mathsf{end}) + (\gamma_4.\ \mathbf{t}'))$ (if $\mathbf{t} \neq \mathbf{t}'$)

▶ **Example 6** (Dynamic Recursive Pipeline). Consider again the dynamic pipeline of Figure 3:

$$\mathit{Pipe} = \lambda\langle \mathbf{p}; \nu\mathbf{q}\rangle.\mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow \mathit{Pipe}\langle \mathbf{q}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})$$

A set of processes that runs according to this specification would proceed as follows. The first iteration is the same as the first iteration of *Pipe*, but without updatable recursion. This is equivalent to the following global type:

$$G_0 = \mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{t}) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})$$

I.e. participant $\mathbf{p}$ would start by sending put or quit to $\mathbf{q}$, and $\mathbf{q}$ would receive this message. Subsequent iterations will *combine* G_0, with the result of the protocol call ($\mathit{Pipe}\langle\mathbf{q}\rangle$). Given a fresh participant $\mathbf{r}$, this is as follows:

$$\begin{aligned} G_1 &= G_0 \between \mathit{Pipe}(\mathbf{q};\mathbf{r}) = G_0 \between (\mu\mathbf{t}.(\mathbf{q} \to \mathbf{r}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow \mathit{Pipe}\langle\mathbf{r}\rangle)) + (\mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end})) \\ &= \mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{q} \to \mathbf{r}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow \mathit{Pipe}\langle\mathbf{r}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end}) \end{aligned}$$

Note that $\between$ plugs in the interactions of the first (second) branch of $\mathit{Pipe}(\mathbf{q};\mathbf{r})$ after the first (resp. second) branch of G_0. This has the effect that, after each iteration of the protocol, a new participant will join the pipeline, until the first participant sends message quit. Such protocols could not be represented in previous MPST extensions. See §6 for a discussion.

▶ **Example 7** (Dynamic Ring). DMst can also be used to model protocols, such a *dynamic ring*, in which participants join a recursive ring protocol. Such dynamic rings are at the core of some well-known protocols, such as Chord and its extensions. The protocol in DMst is as follows, omitting choices and payload types for simplicity:

$$\mathit{Ring} = \lambda\langle \mathbf{i}, \mathbf{p}; \nu\mathbf{q}\rangle.\mu\mathbf{t}.\mathbf{p} \to \mathbf{q}{:}\mathsf{N}.\ \mathbf{t} \blacklozenge (\mathbf{q} \to \mathbf{i}{:}\mathsf{N}.\ \mathbf{i} \hookrightarrow \mathit{Ring}\langle \mathbf{i}, \mathbf{q}\rangle)$$

The entrypoint is $\mathit{Ring}(\mathbf{p},\mathbf{p};\mathbf{q})$. Subsequent iterations would be combined with new protocol calls (e.g. $\mathit{Ring}(\mathbf{p},\mathbf{q};\mathbf{r})$), producing the following sequences of interactions:

$$G_0 = \mu\mathbf{t}.\mathbf{p} \to \mathbf{q}{:}\mathsf{N}.\ \mathbf{q} \to \mathbf{p}{:}\mathsf{N}.\ \mathbf{t} \qquad G_1 = \mu\mathbf{t}.\mathbf{p} \to \mathbf{q}{:}\mathsf{N}.\ \mathbf{q} \to \mathbf{q}'{:}\mathsf{N}.\ \mathbf{q}' \to \mathbf{p}{:}\mathsf{N}.\ \mathbf{t} \qquad G_2 = \ldots$$

3.2 Asynchronous Semantics of DMst Global Types

We guarantee the processes implementing all roles in a global type G indeed behave as G. To characterise the set of behaviours that are allowed by G, we define the semantics of global types as a Labelled State Transition System. The labels are the ***observable actions***:

$$\alpha ::= \mathbf{pq}!\ell \mid \mathbf{pq}?\ell \mid \mathbf{pq}\ \nu\ i@x(\vec{\mathbf{r}};\vec{\mathbf{s}})$$

Observable $\mathbf{pq}!\ell$ is a ***send*** action from $\mathbf{p}$ to $\mathbf{q}$ with an *extended label* (either a label or a protocol call, see Definition 2). Action $\mathbf{pq}?\ell$ is ***receive*** and action $\mathbf{pq}\ \nu\ i@x(\vec{\mathbf{r}};\vec{\mathbf{s}})$ is ***participant creation*** which brings in $\mathbf{q}$ as a new participant acting as the i-th role in the protocol specified by $x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$. For simplicity, we sometimes write $\mathbf{pq}\ \nu\ \ell$ to refer to participant creation, assuming that ℓ is of the form $i@x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$, for some i, x, $\vec{\mathbf{r}}$ and $\vec{\mathbf{s}}$.

Extended Global Types. We extend the global types (Definition 1) with constructs that capture intermediate states of the execution of a protocol. Note that these intermediate states only appear as a result of applying the rules of the operational semantics, and these will not need to be written by users specifying full protocols. Since extended global types are a superset of Definition 1, we will use the same meta-variable G for both and, unless we specify otherwise, all global types from now on are considered to be extended.

$$\gamma ::= \ldots \mid \mathbf{p} \to \mathbf{q}{:}\ell[U] \mid \mathbf{p} \rightsquigarrow \mathbf{q}{:}\ell[U] \mid \mathbf{p}\nu(\vec{\mathbf{r}} : [i,j]@x(\vec{\mathbf{p}}; \vec{\mathbf{q}})) \mid \triangleright[G]$$

Sending protocol call labels (e.g. $\mathbf{p} \to \mathbf{q}{:}i@x(\vec{\mathbf{p}}; \vec{\mathbf{q}})$) is a form of delegation that is used to perform protocol calls (see **Notations** below). $\mathbf{p} \rightsquigarrow \mathbf{q}{:}\ell[U]$ means $\mathbf{p}$ has sent a message to $\mathbf{q}$, yet $\mathbf{q}$ has not received it. $\mathbf{p}\nu(\vec{\mathbf{r}} : [i,j]@x(\vec{\mathbf{p}}; \vec{\mathbf{q}}))$ represents that $\mathbf{p}$ *creates* new participants $\vec{\mathbf{r}}$, acting as the ith to jth participants in x, and $\triangleright[G]$ is the nested protocol with global type G. Intuitively, a nested protocol prefix $\triangleright[G_0].\ G_1$ is equivalent to sequencing G_0 and G_1.

Notations. We use notations to break down protocol calls into the individual interactions. Suppose that $\vec{\mathbf{q}} = (\mathbf{q}_1, \ldots, \mathbf{q}_n)$ and $\vec{\mathbf{r}} = (\mathbf{r}_1, \ldots, \mathbf{r}_m)$. We define $\mathsf{idx}(\mathbf{p}; \vec{\mathbf{q}})$ as $\{i\}$, if $\mathbf{p} = \mathbf{q}_i$ with $1 \le i \le n$, or the empty set $\{\}$ otherwise. We define the following shortcuts:

$$\begin{aligned} \mathbf{p} \to \vec{\mathbf{q}}{:}\vec{i}@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) &= \mathbf{p} \to \mathbf{q_1}{:}i_1@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}).\ \ldots.\ \mathbf{p} \to \mathbf{q_n}{:}i_n@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) \\ \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) &= \mathbf{p}\nu(\vec{\mathbf{r}} : [n+1, n+m]@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})).\ \mathbf{p} \to \vec{\mathbf{q}} \setminus \{\mathbf{p}\}{:}([1,n] \setminus \mathsf{idx}(\mathbf{p}; \vec{\mathbf{q}}))@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) \end{aligned}$$

Notation $\mathbf{p} \to \vec{\mathbf{q}}{:}\vec{i}@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$ represents a sequence of messages from $\mathbf{p}$ to each of the $\mathbf{q} \in \vec{\mathbf{q}}$ with the respective extended label. These are sometimes called **invitations** to x. Notation $\mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$ is a sequence of actions, where $\mathbf{p}$ first creates $\vec{\mathbf{r}}$, and then sends invitations to $\vec{\mathbf{q}}$, excluding itself to avoid self-communication.

Global Type Equivalence and LTS. We define the erasure of updatable recursive variables as $|\mathbf{t}\blacklozenge\gamma|_{\mathbf{t}'} = \mathbf{t}$ if $\mathbf{t} = \mathbf{t}'$; and $|\mathbf{t}\blacklozenge\gamma|_{\mathbf{t}'} = \mathbf{t}\blacklozenge\gamma$ otherwise (other cases are homomorphic). The LTS is defined up to the equivalence: (1) $\mathbf{p}\nu(); G \equiv G$; (2) $\triangleright[\mathsf{end}].\ G \equiv G$; (3) $\mu\mathbf{t}.G \equiv [\mu\mathbf{t}.|G|_{\mathbf{t}}/\mathbf{t}]G$, and, assuming $\vec{\mathbf{r}}$ fresh, (4) $\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle.\ G \equiv \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}}; \vec{\mathbf{r}}).\ \triangleright[x(\vec{\mathbf{q}}; \vec{\mathbf{r}})].\ G$, and (5) $G \blacklozenge (\vec{\gamma}.\ \mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle) \equiv \vec{\gamma}.\ \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}}; \vec{\mathbf{r}}).\ (G \lozenge x(\vec{\mathbf{q}}; \vec{\mathbf{r}}))$. Rules (1) and (2) capture that finished prefixes (creating an empty list of participants, or a nested ended global type) can be skipped. Rule (3) is recursion unrolling. Similarly to Example 6, subsequent iterations of the protocol will combine the body of the recursion without updatable recursion variables, with the result of the protocol calls. By this rule, recursion will be updated by protocol calls, and after the first iteration, the protocol can continue as $\mu\mathbf{t}.|G|_{\mathbf{t}}$ (possibly combined with the result of a protocol call). Rule (4) expands the sequence of a protocol call and a global type, and rule (5) *updates* a global type by first executing the specified prefixes and then continuing with G combined with the result of the protocol call. We guarantee that new roles are globally fresh by adopting a Barendregt convention on all binders, i.e. each time we access a protocol definition x, we alpha-rename the participants bound by $\nu\vec{\mathbf{r}}$ to avoid participant name clashes. Without it, consecutive protocol calls could incorrectly introduce repeated participant names.

▶ **Definition 8** (Active Participants). *The active participants of a global type (prefix),* $\mathsf{pt}(G)$ *(*$\mathsf{pt}(\gamma)$*), is the set of participants that can perform an action in the protocol (or prefix).*

$$\mathsf{pt}(\mathbf{p} \to \mathbf{q}{:}\ell[U]) = \{\mathbf{p}, \mathbf{q}\} \quad \mathsf{pt}(\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle) = \{\mathbf{p}\} \cup \vec{\mathbf{q}} \quad \mathsf{pt}(\mathbf{p} \rightsquigarrow \mathbf{q}{:}\ell[U]) = \{\mathbf{q}\} \quad \gamma.G = \mathsf{pt}(\gamma) \cup \mathsf{pt}(G)$$
$$\mathsf{pt}(\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle) = \{\mathbf{p}\} \cup \vec{\mathbf{q}} \quad \mathsf{pt}(\mathbf{p}\nu(\vec{\mathbf{r}} : [i,j]@x(\vec{\mathbf{p}}; \vec{\mathbf{q}}))) = \{\mathbf{p}\} \cup \vec{\mathbf{r}} \quad \mathsf{pt}(\triangleright[G]) = \mathsf{pt}(G)$$

$$\mathsf{pt}(\mathsf{end}) = \mathsf{pt}(\mathbf{t}) = \{\} \quad \mathsf{pt}(\mu\mathbf{t}.G) = \mathsf{pt}(G) \quad \mathsf{pt}(\textstyle\sum_{i\in I} G_i) = \bigcup_{i\in I} \mathsf{pt}(G_i) \quad \mathsf{pt}(G \blacklozenge \gamma) = \mathsf{pt}(G) \cup \mathsf{pt}(\gamma)$$

▶ **Definition 9** (LTS for Global Types). Let the ***subject*** of an action denote the role that performs it: $\mathbf{p} = \mathsf{subj}(\mathbf{pq}!\ell) = \mathsf{subj}(\mathbf{pq}?\ell) = \mathsf{subj}(\mathbf{pq}\ \nu\ \ell)$. The LTS for G:

$$[\textsc{Br-A}]\ \frac{\forall i \in I, G_i \xrightarrow{\alpha} G'_i}{\sum_{i\in I} G_i \xrightarrow{\alpha} \sum_{i\in I} G'_i} \qquad [\textsc{Br-B}]\ \frac{G_j \xrightarrow{\mathbf{pq}!\ell_j} G'_j \quad \mathtt{dc}(\mathbf{p}, \{\ell_i\}_i, \sum_{i\in I} G_i)}{\sum_{i\in I} G_i \xrightarrow{\mathbf{pq}!\ell_j} G'_j} \qquad [\textsc{Nest}]\ \frac{G_1 \xrightarrow{\alpha} G_2}{\triangleright[G_1].\ G \xrightarrow{\alpha} \triangleright[G_2].\ G}$$

$$[\textsc{New}]\ \mathbf{p}(\mathbf{r}, \vec{\mathbf{r}} : [i,j]@x(\vec{\mathbf{q}};\vec{\mathbf{r}}')).\ G \xrightarrow{\mathbf{pr}\ \nu\ i@x(\vec{\mathbf{q}};\vec{\mathbf{r}}')} \mathbf{p}(\vec{\mathbf{r}} : [i+1,j]@x(\vec{\mathbf{q}};\vec{\mathbf{r}}')).\ G$$

$$[\textsc{Send}]\ \mathbf{p} \to \mathbf{q}{:}\ell[U].\ G \xrightarrow{\mathbf{pq}!\ell} \mathbf{p} \rightsquigarrow \mathbf{q}{:}\ell[U].\ G \qquad [\textsc{Recv}]\ \mathbf{p} \rightsquigarrow \mathbf{q}{:}\ell[U].\ G \xrightarrow{\mathbf{qp}?\ell} G \qquad [\textsc{Seq}]\ \frac{G \xrightarrow{\alpha} G' \quad \mathsf{subj}(\alpha) \notin \mathsf{pt}(\gamma)}{\gamma.G \xrightarrow{\alpha} \gamma.G'}$$

[Br-A] specifies that if an action can be taken in all branches of a choice, it can be taken before the choice is decided. The reason is that if an action can be taken in all branches, then it must be *independent* of the choice. [Br-B] states that if the sender of a choice does an action that selects branch j, then the choice transitions to this branch. [Seq] states that an action can take place in a continuation, if the action does not involve the participants of the prefix. In the prefix transitions, [Send] (resp. [Recv]) represents a send (resp. receive) action. [New] specifies that a new participant $\mathbf{r}$ of the nested protocol is created, and [Nest] represents the execution of an action in the nested global type.

▶ **Example 10** (DMst Semantics)**.** Consider the following protocol, cf. Example 6:

$$Pipe = \lambda\langle \mathbf{p}; \nu\mathbf{q}\rangle.\mu\mathbf{t}.G_0 \quad \text{with } G_0 = (\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})$$

First, assuming two initial participants ($\mathbf{p}$ and $\mathbf{q}$), we unfold recursion using $\equiv$:

$$\mu\mathbf{t}.G_0 \equiv [\mu\mathbf{t}.|G_0|_\mathbf{t}/\mathbf{t}]G_0 = (\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mu\mathbf{t}.|G_0|_\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})$$

There are two allowed actions: sending put and sending quit. By [Br-B] and [Send],

$$[\mu\mathbf{t}.|G_0|_\mathbf{t}/\mathbf{t}]G_0 \xrightarrow{\mathbf{pq}!\mathsf{put}} \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mu\mathbf{t}.|G_0|_\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle)$$

There are now two actions accepted. First, we can use [Recv]:

$$\mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mu\mathbf{t}.|G_0|_\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle) \xrightarrow{\mathbf{qp}?\mathsf{put}} \mu\mathbf{t}.|G_0|_\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle$$

To enable the second action, we use equivalences to unfold the updatable global type:

$$\begin{aligned} G_1 &= \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mu\mathbf{t}.|G_0|_\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle\mathbf{q}\rangle) \\ &\equiv \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{q}\nu(\mathbf{r} : 2@Pipe(\mathbf{q};\mathbf{r})).\ (\mu\mathbf{t}.|G_0|_\mathbf{t} \between Pipe(\mathbf{q};\mathbf{r}))\end{aligned}$$

Note that $\mathbf{p}$ is not in the set of active participants of the prefix, so $\mathbf{p}$ can take a step, using repeated applications of [Seq], in $(\mu\mathbf{t}.|G_0|_\mathbf{t} \between Pipe(\mathbf{q};\mathbf{r}))$.

$$\begin{aligned} &\mu\mathbf{t}.G_2 = (\mu\mathbf{t}.|G_0|_\mathbf{t} \between Pipe(\mathbf{q};\mathbf{r})) \\ &= \mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{q} \to \mathbf{r}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow Pipe\langle\mathbf{r}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end}) \\ &\equiv (\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{q} \to \mathbf{r}{:}\mathsf{put}[\mathsf{nat}].\ (\mu\mathbf{t}.|G_2|_\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow Pipe\langle\mathbf{r}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end})\end{aligned}$$

Suppose that G_2 proceeds by $\mathbf{p}$ sending quit: $\mu\mathbf{t}.G_2 \xrightarrow{\mathbf{pq}!\mathsf{quit}} (\mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end})$. Then, G_1 transitions to the following global type:

$$\mu\mathbf{t}.G_1 \xrightarrow{\mathbf{pq}!\mathsf{quit}} \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ \mathbf{q}\nu(\mathbf{r} : 2@Pipe(\mathbf{q};\mathbf{r})).\ \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end}$$

After [Sq-A] and [Recv], the global type transitions to:

$$G_3 = \mathbf{q}\nu(\mathbf{r} : 2@Pipe(\mathbf{q};\mathbf{r})).\ \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end}$$

With [Sq-A] and [New], the protocol transitions as follows:

$$G_3 \xrightarrow{\mathbf{qr}\ \nu\ 2@Pipe(\mathbf{q};\mathbf{r})} \mathbf{p} \rightsquigarrow \mathbf{q}{:}\mathsf{quit}.\ \mathbf{q} \to \mathbf{r}{:}\mathsf{quit}.\ \mathsf{end}$$

At this stage, $\mathbf{r}$ is a new active participant of the protocol. The remaining global type can run to completion via a sequence of [Recv], [Send], and finally [Recv].

3.3 Local Types

Local types describe the interactions of a protocol from the point of view of a single participant.

▶ **Definition 11** (DMst Local Types). *Let* $M ::= l[U] \mid L$. *The syntax of local types is:*
$\pi ::= \mathbf{p}!M \mid \mathbf{p}?M \mid \nu(\vec{\mathbf{p}} : \vec{L}) \mid \triangleright[L] \qquad L ::= \mathsf{end} \mid \pi.\ L \mid \sum_{i\in I} L_i \mid \mu\mathbf{t}.L \mid \mathbf{t} \mid L \blacklozenge \vec{\pi}$

Local type syntax differs from that of global types in the prefixes (π instead of γ). Local type prefixes are as follows: ***send*** $\mathbf{p}!M$, ***receive*** $\mathbf{p}?M$, ***new participant creation*** $\nu(\mathbf{p}_1 : L_1)\cdots(\mathbf{p}_n : L_n)$, and the ***nested local type*** $\triangleright[L]$. We lift the definitions of directed choices, updatable recursion erasure, and the composition operator from global types to local types. ***Endpoint projection*** takes a global type G and a participant $\mathbf{r}$, and produces the local type (the local interactions) of $\mathbf{r}$ in G.

Similarly to global types, we introduce the notations for protocol calls. Assuming $\vec{\mathbf{q}} = \mathbf{q}_1, \ldots, \mathbf{q}_n$ and $\vec{\mathbf{r}} = \mathbf{r}_1, \ldots, \mathbf{r}_m$, we define these notations as follows:

$$\begin{aligned} \vec{\mathbf{q}}!\vec{i}@x(\vec{\mathbf{q}};\vec{\mathbf{r}}) &= \mathbf{q_1}!i_1@x(\vec{\mathbf{q}};\vec{\mathbf{r}}).\ \ldots.\ \mathbf{q_n}!i_n@x(\vec{\mathbf{q}};\vec{\mathbf{r}}) \\ \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}};\vec{\mathbf{r}}) &= \nu(\vec{\mathbf{r}} : [n+1, n+m]@x(\vec{\mathbf{q}};\vec{\mathbf{r}})).\ \vec{\mathbf{q}} \setminus \mathbf{p}!([1,n] \setminus \mathsf{idx}(\mathbf{p};\vec{\mathbf{q}}))@x(\vec{\mathbf{q}};\vec{\mathbf{r}}) \end{aligned}$$

▶ **Definition 12** (Prefix Projection). *Global type projection is defined in terms of prefix projection. Prefix projection is a partial function that takes a global prefix, and produces a possibly empty (ε) sequence of local prefixes. We give the two main rules:*

$$\begin{aligned} &\mathbf{p} \to \mathbf{q}{:}l[U] \restriction \mathbf{r} \\ &= \begin{cases} \mathbf{q}!l[U] & \mathbf{p} = \mathbf{r} \neq \mathbf{q} \\ \mathbf{p}?l[U] & \mathbf{p} \neq \mathbf{r} = \mathbf{q} \\ \varepsilon & \mathbf{p}, \mathbf{r}, \mathbf{q}\ \textit{distinct} \end{cases} \end{aligned} \qquad \begin{aligned} &\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle \restriction \mathbf{r} \qquad\qquad (\vec{\mathbf{r}}\ \textit{fresh}) \\ &= \begin{cases} \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}};\vec{\mathbf{r}}).\ \triangleright[i@x(\vec{\mathbf{q}};\vec{\mathbf{r}})] & \mathbf{p} = \mathbf{r} \in_i \vec{\mathbf{q}} \\ \mathbf{p}\ \mathsf{call}\ x(\vec{\mathbf{q}};\vec{\mathbf{r}}) & \mathbf{p} = \mathbf{r} \notin \vec{\mathbf{q}} \\ \mathbf{p}?i@x(\vec{\mathbf{q}};\vec{\mathbf{r}}).\ \triangleright[i@x(\vec{\mathbf{q}};\vec{\mathbf{r}})] & \mathbf{p} \neq \mathbf{r} \in \vec{\mathbf{q}} \\ \varepsilon & \mathbf{p} \neq \mathbf{r} \notin \vec{\mathbf{q}} \end{cases} \end{aligned}$$

The projection of $\mathbf{p} \to \mathbf{q}{:}l[U]$ onto $\mathbf{r}$ is a *send* if $\mathbf{r}$ is $\mathbf{p}$, and a *receive* if $\mathbf{r}$ is $\mathbf{q}$, an empty prefix if all roles are distinct, or undefined if $\mathbf{r} = \mathbf{p} = \mathbf{q}$. The projection of $\mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle$ follows a similar pattern. If $\mathbf{r}$ is $\mathbf{p}$, then the projected sequence of prefixes is the one that corresponds to making the protocol call, i.e. delegating channels and creating new participants. If $\mathbf{r}$ is the ith participant in $\vec{\mathbf{q}}$, then $\mathbf{r}$ also takes part in the protocol, so the prefixes correspond to the reception of the channel for acting as the ith participant in x, followed by the execution of the nested local type for this ith participant in x, $i@x(\vec{\mathbf{q}};\vec{\mathbf{r}})$. If $\mathbf{r}$ is both the protocol caller $\mathbf{p}$, and also takes part in it, then the prefix sequence is the sequence of prefixes for making the protocol call, followed by the nested local type for $i@x$. Note that a participant may call a protocol, and not take part in it. When this happens, the protocol caller simply distributes the necessary channels for executing the nested protocol, and then proceeds to the continuation, without entering the nested protocol.

▶ **Definition 13** (Projection and Merging). Projection is defined as follows:

$$\gamma.G \restriction \mathbf{r} = \gamma \restriction \mathbf{r}.\ G \restriction \mathbf{r} \qquad \begin{aligned} \mathbf{t} \restriction \mathbf{r} &= \mathbf{t} \\ \mathsf{end} \restriction \mathbf{r} &= \mathsf{end} \end{aligned} \qquad G \blacklozenge \vec{\gamma} \restriction \mathbf{r} = \begin{cases} G \restriction \mathbf{r} & (\text{if } \mathbf{r} \notin \vec{\gamma}) \\ G \restriction \mathbf{r} \blacklozenge (\vec{\gamma} \restriction \mathbf{r}) & (\text{if } \mathbf{r} \in \vec{\gamma}) \end{cases}$$

$$\mu\mathbf{t}.G \restriction \mathbf{r} = \begin{cases} \mu\mathbf{t}.G \restriction \mathbf{r} & \mathbf{r} \in \mathsf{pt}(G) \text{ or } \\ & \text{fv}(\mu\mathbf{t}.G) \neq \emptyset \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad \sum_{i\in I} G_i \restriction \mathbf{r} = \begin{cases} \sum_{i\in I} (G_i \restriction \mathbf{r}) & \mathtt{dc}(\mathbf{p}, \vec{\ell}, \sum_{i\in I} G_i), \mathbf{r} = \mathbf{p} \\ \bigsqcap_{i\in I} (G_i \restriction \mathbf{r}) & \mathtt{dc}(\mathbf{p}, \vec{\ell}, \sum_{i\in I} G_i), \mathbf{r} \neq \mathbf{p} \end{cases}$$

Projection is a partial function from global to local types. We lift the definition of directed choices (Definition 3, dc) to local types. We define $\bigsqcap_{i\in I} L_i$ as the *merging* operator:

(1) $L \sqcap L = L$ (2) $\mu\mathbf{t}.L_1 \sqcap \mu\mathbf{t}.L_2 = \mu\mathbf{t}.(L_1 \sqcap L_2)$

(3) $\sum_{i\in I} \mathbf{p}?\ell_i[U_i].\ L_i \sqcap \sum_{j\in J} \mathbf{p}?\ell_j[U_j].\ L'_j =$
$\sum_{k\in I\cap J} (\mathbf{p}?\ell_k[U_k].\ L_k \sqcap L'_k) + \sum_{i\in I\setminus J} (\mathbf{p}?\ell_i[U_i].\ L_i) + \sum_{j\in J\setminus I} (\mathbf{p}?\ell_j[U_j].\ L'_j)$

The projection rules are standard [60], except the choice. A choice is only defined if it is directed. The projection of the participant that makes the choice is a local type choice of the projection of the branches. The projection for all other participants is the *merging* of the projection of the branches. Local types can be merged in three cases: (1) they are the same, (2) they are recursive local types whose bodies can be merged, or (3) they become aware of which branch of the choice was taken (if necessary), by receive actions with distinct labels. Case (3) implies that both local types are choices with a receive prefix as the first action, where the continuations for the branches with the same labels can be merged.

It is standard in MPST to define *well-formedness* in terms of *projectability* [21]. This means that if a global type is projectable onto all of its roles, then it is well-formed and therefore live and deadlock-free. Unfortunately, the use of $\blacklozenge$ means that this is not possible with DMst. E.g., the following global type is projectable, but it will get stuck:

$$\begin{aligned}&\mathit{Proto1} = \lambda\langle \mathbf{p}; \nu\mathbf{r}\rangle.\mu\mathbf{t}.(\mathbf{r} \to \mathbf{p}{:}m_1.\ \mathsf{end}) + (\mathbf{r} \to \mathbf{p}{:}m_2.\ \mathbf{t})\\ &\mathit{IllFormed} = \lambda\langle \mathbf{p}; \nu\mathbf{q}\rangle.\mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}m_1.\ \mathsf{end}) + (\mathbf{p} \to \mathbf{q}{:}m_2.\ \mathbf{t} \blacklozenge (\mathbf{p} \hookrightarrow \mathit{Proto1}\langle\mathbf{p}\rangle))\end{aligned}$$

Specifically, $\mathbf{r}$ in *Proto1* will not become aware of the branch taken by $\mathbf{p}$ in *IllFormed*, so after unfolding *IllFormed* once, we will obtain the following global type:

$$\mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}m_1.\ \mathbf{r} \to \mathbf{p}{:}m_1.\ \mathsf{end}) + (\mathbf{p} \to \mathbf{q}{:}m_2.\ \mathbf{r} \to \mathbf{p}{:}m_2.\ \mathbf{t})$$

But this protocol would not be projectable. To avoid such cases, we define a necessary condition for well-formedness, the *safe protocol update* condition.

▶ **Definition 14** (Safe Protocol Update). *Suppose that $C[\ ]$ and $C'[\ ]$ are 1-hole global type contexts. A global type $\mu\mathbf{t}.C[\mathbf{t} \blacklozenge (\vec{\gamma}.\ \mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle)]$ contains a safe update if its 1-unfolding is some $C'[G \blacklozenge (\vec{\gamma}.\ \mathbf{p} \hookrightarrow x\langle\vec{\mathbf{q}}\rangle)]$, such that given a sequence of fresh roles $\vec{\mathbf{r}}$, $G \lozenge x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$ is projectable.*

▶ **Definition 15** (Projection and Well-Formed Global Types). *A global type G is projectable if its projection $G \upharpoonright \mathbf{r}$ is defined on all roles $\mathbf{r} \in G$. A global type is well formed iff it is projectable, and contains only safe protocol updates.*

▶ **Definition 16** (Projections of Protocol Definitions). *Assume a definition $x = \lambda\langle\vec{\mathbf{p}}; \nu\vec{\mathbf{p}}'\rangle.G$, with participants $\vec{\mathbf{p}} = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$ and with participants $\vec{\mathbf{p}}' = (\mathbf{p}_{n+1}, \ldots, \mathbf{p}_m)$. The projections of x are the local protocol definitions that correspond to each of the participants in the protocol:*

$$1@x = \lambda\langle\vec{\mathbf{p}}; \nu\vec{\mathbf{p}}'\rangle.G \upharpoonright \mathbf{p}_1 \quad \ldots \quad m@x = \lambda\langle\vec{\mathbf{p}}; \nu\vec{\mathbf{p}}'\rangle.G \upharpoonright \mathbf{p}_m$$

▶ **Example 17** (Directed Choices and Merging). *BFib* computes the n-th Fibonacci number:

$$\begin{aligned}&\mathit{BFib} = \lambda\langle \mathbf{r}, \mathbf{f_1}, \mathbf{f_2}; \nu\mathbf{f_3}\rangle.\mathbf{f_1} \to \mathbf{f_3}{:}\mathsf{F}[\mathsf{int}].\\ &\qquad \mathbf{f_2} \to \mathbf{f_3}{:}\mathsf{F}[\mathsf{int}].((\mathbf{f_3} \to \mathbf{r}{:}\mathsf{NF}[\mathsf{int}].\mathbf{f_3} \to \mathbf{f_2}{:}\mathsf{quit}.\mathsf{end}) + (\mathbf{f_3} \hookrightarrow \mathit{BFib}\langle\mathbf{r}, \mathbf{f_2}, \mathbf{f_3}\rangle.\ \mathsf{end}))\end{aligned}$$

This protocol is similar to that of Example 4, but instead of calling *BFib* indefinitely, the protocol offers a choice: $\mathbf{f_3}$ will either reply to $\mathbf{r}$ with its Fibonacci number, or call *BFib* recursively to compute the next number. Participant $\mathbf{f_3}$ *selects* the branch of the protocol that is taken, and $\mathbf{r}$ *offers* the two branches. The choice has a single sender, and both branches can be distinguished by the labels or protocol calls, so the choice is *directed* by $\mathbf{f_3}$, with extended labels $\vec{\ell} = \mathsf{NF}, i@\mathit{BFib}(\mathbf{r}, \mathbf{f_2}, \mathbf{f_3}; \mathbf{f_4})$. In a directed choice, one participant decides the branch. But how do the remaining participants *know* which branch was taken? Consider $\mathbf{f_1}$ in *BFib*. Its part in both branches of the protocol is the same, end, so we can project $\mathbf{f_1}$ in the choice as end. This is one of the cases of Definition 13: two local types can be merged if they are the same. But $\mathbf{f_2}$'s behaviour is different in each branch: $\mathbf{f_3}?\mathsf{quit}.\ \mathsf{end}$ and $\mathbf{f_3}?2@\mathit{BFib}(\mathbf{r}, \mathbf{f_2}, \mathbf{f_3}; \mathbf{f_4}).\ \mathsf{end}$ respectively. However, $\mathbf{f_2}$ is *aware* of the branch that was taken by receiving either label quit or protocol call label $2@\mathit{BFib}(\mathbf{r}, \mathbf{f_2}, \mathbf{f_3}; \mathbf{f_4})$. This is case (3), as explained after Definition 13:

$$(\mathsf{f_3}?\mathsf{quit}.\ \mathsf{end}) \sqcap (\mathsf{f_3}?2@\mathit{BFib}(\mathbf{r}, \mathsf{f_2}, \mathsf{f_3}; \mathsf{f_4}).\ \mathsf{end}) = (\mathsf{f_3}?\mathsf{quit}.\ \mathsf{end}) + (\mathsf{f_3}?2@\mathit{BFib}(\mathbf{r}, \mathsf{f_2}, \mathsf{f_3}; \mathsf{f_4}).\ \mathsf{end})$$

▶ **Example 18** (Projecting Pipeline). Consider again Example 6. We are projecting the first and second participants of x. The result of the syntactic projection is as follows:

$$\begin{aligned}
x(\mathbf{p}; \mathbf{q}) &= \mu\mathbf{t}.(\mathbf{p} \to \mathbf{q}{:}\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow x\langle\mathbf{q}\rangle)) + (\mathbf{p} \to \mathbf{q}{:}\mathsf{quit}.\ \mathsf{end})\\
1@x(\mathbf{p}; \mathbf{q}) &= \mu\mathbf{t}.(\mathbf{q}!\mathsf{put}[\mathsf{nat}].\ \mathbf{t}) + (\mathbf{q}!\mathsf{put}[\mathsf{nat}].\ \mathsf{end})\\
2@x(\mathbf{p}; \mathbf{q}) &= \mu\mathbf{t}.(\mathbf{p}?\mathsf{put}[\mathsf{nat}].\ (\mathbf{t} \blacklozenge (\mathsf{call}\ x(\mathbf{q}; \mathbf{r}).\ \mathbf{q}?\mathbf{q}1@x(\mathbf{q}; \mathbf{r}).\ \triangleright[1@x(\mathbf{q}; \mathbf{r})]))) + (\mathbf{p}?\mathsf{quit}.\ \mathsf{end})
\end{aligned}$$

3.4 Semantics of DMst Local Types and Correctness

The semantics for local types is defined for ***local type configurations***. A configuration is a pair of *channel* and *participant* environments, $\langle\ \Delta\ ;\ \Theta\ \rangle$. The channel environment Δ contains the shared channels used for the asynchronous communication between each pair of participants, and the participant environment Θ is a set of the local types of all participants:

$$\Delta = \mathsf{p_i q_j} :: \vec{\mathsf{w}}_1, \ldots, \mathsf{p_k q_l} :: \vec{\mathsf{w}}_n \qquad \mathsf{w} ::= \ell[U] \qquad \Theta = \{\mathsf{p_1} :: L_1, \cdots, \mathsf{q_m} :: L_m\}$$

w denotes a *payload* of a message. We consider the channel and participant environments up to commutativity and associativity, since all entries must be disjoint. Channels $\mathbf{pq}$ are channels of messages to $\mathbf{p}$ from $\mathbf{q}$. We use $\Delta(\mathbf{pq})$ as notation for retrieving channel $\mathbf{pq}$, and $\Delta[\mathbf{pq} :: \vec{\mathbf{w}}]$ for updating channel $\mathbf{pq}$ with $\vec{\mathbf{w}}$. Θ does not impose the ordering between the entries (like a set). We update the entry by writing $\Theta[\mathbf{p} :: L] = \mathbf{p} :: L, (\Theta \setminus \mathbf{p})$.

The semantics of configurations is defined by the LTS of local types and given in Definition 19, and it is defined up to local type equivalences, analogous to those of global types: (1) $\mu\mathbf{t}.L \equiv \vec{\pi}.\ \mathsf{end}$ if $L = \mathbf{t} \blacklozenge \vec{\pi}$; (2) $[\mu\mathbf{t}.|L|_{\mathbf{t}}/\mathbf{t}]L$ if $L \neq \mathbf{t} \blacklozenge \vec{\pi}$, (3) $L \blacklozenge (\vec{\pi}.\ \pi) \equiv \vec{\pi}.\ \pi.\ L$, if $\pi \neq \triangleright[L']$, and (4) $L \blacklozenge (\vec{\pi}.\ \pi) \equiv \vec{\pi}.\ (L \lozenge L')$, if $\pi = \triangleright[L']$. The semantics of choices requires that they are directed. At the local type, all branches start with a send/receive prefix to/from the same participant $\mathbf{p}$. We use the predicate $\mathtt{dc}(\mathbf{p}, \{\ell_i\}_i, \sum_{i\in I} \pi_i.\ L_i)$, and define it analogously to the predicate for global types.

▶ **Definition 19** (LTS for Local Types). The LTS for local types is defined as follows:

$$[\text{L-CONG}]\ \frac{\langle\ \Delta\ ;\ \mathbf{p} :: L\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \Theta'\ \rangle}{\langle\ \Delta\ ;\ \mathbf{p} :: L, \Theta\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \Theta', \Theta\ \rangle} \qquad [\text{L-NEST}]\ \frac{\langle\ \Delta\ ;\ \mathbf{p} :: L'\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \mathbf{p} :: L'', \Theta\ \rangle}{\langle\ \Delta\ ;\ \mathbf{p} :: \triangleright[L'].\ L\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \mathbf{p} :: \triangleright[L''].\ L, \Theta\ \rangle}$$

$$[\text{L-CHOICE}]\ \frac{j \in I \quad \langle\ \Delta\ ;\ \mathbf{p} :: L_j\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \Theta\ \rangle \quad \mathtt{dc}(\mathbf{q}, \vec{\ell}, \sum_{i\in I} L_i)}{\langle\ \Delta\ ;\ \mathbf{p} :: \sum_{i\in I} L_i\ \rangle \xrightarrow{\alpha} \langle\ \Delta'\ ;\ \Theta\ \rangle}$$

$$[\text{L-SEND}]\ \langle\ \Delta, \mathbf{qp} :: \vec{\mathbf{w}}\ ;\ \mathbf{p} :: \mathbf{q}!\ell[U].\ L\ \rangle \xrightarrow{\mathbf{pq}!\ell} \langle\ \Delta, \mathbf{qp} :: \vec{\mathbf{w}} \cdot \ell[U]\ ;\ \mathbf{p} :: L\ \rangle$$

$$[\text{L-RECV}]\ \langle\ \Delta, \mathbf{pq} :: \ell[U] \cdot \vec{\mathbf{w}}\ ;\ \mathbf{p} :: \mathbf{q}?\ell[U].\ L\ \rangle \xrightarrow{\mathbf{pq}?\ell} \langle\ \Delta, \mathbf{pq} :: \vec{\mathbf{w}}\ ;\ \mathbf{p} :: L\ \rangle$$

$$[\text{L-NEW}]\ \langle\ \Delta\ ;\ \mathbf{p} :: \nu(\mathbf{q}_i : L_i)\cdots(\mathbf{q}_j : L_j)\ \rangle \xrightarrow{\mathbf{pq_i}\ \nu\ L_i} \langle\ \Delta\ ;\ \mathbf{q_i} :: L_i, \mathbf{p} :: \nu(\mathbf{q}_{i+1} : L_{i+1})\cdots(\mathbf{q}_j : L_j)\ \rangle$$

[L-CONG] specifies a step by a participant in the configuration. [L-RECUR] unfolds recursion, and [L-CHOICE] selects one branch of a choice by performing a step into one of the continuations. In [L-CHOICE], only one action can take place in one branch, because the labels of all branches must be distinct for the choice to be directed. [L-SEND] executes a send prefix by enqueuing the label and the payload type into the channel of the receiver, [L-RECV] executes a receive prefix by dequeuing the label and payload type from the corresponding channel, [L-NEW] creates a new participant by composing its associated local type in parallel with the remainder of the local type environment, [L-NEST] performs a step into a nested local type. We allow the renaming of participants introduced by [L-NEW] to avoid participant name clashes. For simplicity, we assume that Δ always contains a (possibly empty) sequence of payloads for every pair of roles. For example, if $\mathbf{pq}$ is not in Δ, we allow to match Δ with $\Delta, \mathbf{pq} :: \epsilon$.

We prove the correctness of DMst: (1) the global type semantics coincides with behaviours of local endpoints, a well-formed global type is (2) deadlock-free and (3) live. (1) together with (2) and (3) imply that the programs generated from local types projected from well-formed global types are deadlock-free and live.

We define ***the projection of*** G as $[\![G]\!] = \langle\, [] \,;\, \mathbf{p} :: G \upharpoonright \mathbf{p}, \ldots, \mathbf{q} :: G \upharpoonright \mathbf{q} \,\rangle$, for all $\mathbf{p}, \ldots, \mathbf{q} \in \mathsf{pt}(G)$. A configuration is a subtype of another if it contains the same participants and their local types are related under the standard subtyping relation [60], i.e., $\langle\, \Delta \,;\, \Theta \,\rangle \leqslant \langle\, \Delta \,;\, \Theta' \,\rangle$ implies that $\Theta(\mathbf{p}) \leqslant \Theta'(\mathbf{p})$ for all $\mathbf{p}$.

▶ **Theorem 20** (Trace Equivalence). *If* $\langle\, \Delta \,;\, \Theta \,\rangle \leqslant [\![G]\!]$, *then* $\Gamma \vdash G \xrightarrow{\alpha*} G'$ *if and only if there exists* $\langle\, \Delta' \,;\, \Theta' \,\rangle$ *such that* $\langle\, \Delta \,;\, \Theta \,\rangle \xrightarrow{\alpha*} \langle\, \Delta' \,;\, \Theta' \,\rangle$ *and* $\langle\, \Delta' \,;\, \Theta' \,\rangle \leqslant [\![G']\!]$.

Proof. The full proof uses the extended projection, that produces both local types and the queue contents implicit in the intermediate forms. The core part of the proof is completed by induction on the derivations for the global and local type LTS, using the fact that if $G \equiv G'$, then $G \upharpoonright \mathbf{r} \equiv G' \upharpoonright \mathbf{r}$. ◀

A configuration $\langle\, \Delta \,;\, \Theta \,\rangle$ is *final* if for all $\mathbf{pq} \in \mathrm{dom}(\Delta)$, $\Delta(\mathbf{pq}) = \varepsilon$, and for all $\mathbf{p} \in \mathrm{dom}(\Theta)$, $\Theta(\mathbf{pq}) = \mathsf{end}$. The configuration is in a deadlock if it cannot make progress and it is not final, i.e. the protocol has not ended, and all participants are stuck.

▶ **Definition 21** (Deadlock). $\langle\, \Delta \,;\, \Theta \,\rangle$ is a *deadlock configuration* if there exists a sequence of actions $\alpha*$ such that $\langle\, \Delta \,;\, \Theta \,\rangle \xrightarrow{\alpha*} \langle\, \Delta' \,;\, \Theta' \,\rangle$, with $\langle\, \Delta' \,;\, \Theta' \,\rangle$ not final and for all action α, $\langle\, \Delta' \,;\, \Theta' \,\rangle \not\xrightarrow{\alpha}$.

▶ **Example 22** (Deadlock Configuration). A deadlock configuration is one in which the whole system can get stuck and cannot progress. A usual example of this is a configuration where all participants need to receive, but their messages have not been sent. We show below such configuration, where after one action, it reaches a receive cycle:

$$\begin{array}{l} \langle\, [\qquad\qquad] \,;\, \mathbf{p} :: \mathbf{q}!l[U].\, \mathbf{q}?l[U].\, L_1, \mathbf{q} :: \mathbf{r}?l[U].\, L_2, \mathbf{r} :: \mathbf{p}?l[U].\, L_3 \,\rangle \xrightarrow{\mathbf{pq}!l} \\ \langle\, [\mathbf{qp} :: l[U]] \,;\, \mathbf{p} :: \mathbf{q}?l[U].\, L_1 \qquad\quad , \mathbf{q} :: \mathbf{r}?l[U].\, L_2, \mathbf{r} :: \mathbf{p}?l[U].\, L_3 \,\rangle \not\xrightarrow{\alpha} \end{array}$$

▶ **Theorem 23** (Deadlock-Freedom). *If* $\langle\, \Delta \,;\, \Theta \,\rangle \leqslant [\![G]\!]$, *then* $\langle\, \Delta \,;\, \Theta \,\rangle$ *is deadlock-free.*

Proof. We show that either G is ended, or there is a step available for G, and use trace equivalence to conclude this for $\langle\, \Delta \,;\, \Theta \,\rangle \leqslant [\![G]\!]$. ◀

Theorem 23 refers exclusively to the absence of *global* deadlocks, i.e. the whole system will never get stuck. But DMst also guarantees the absence of *local* deadlocks, i.e. that no participant in the system gets stuck. An example of such partial deadlocks is the usual *receive-cycle*, where a subset of participants are waiting forever, and can never make progress. DMst guarantees that this situation cannot happen. To prove this, we first show that DMst guarantees *orphan message freedom* [10], which means that all messages are eventually consumed without a type mismatch.

▶ **Definition 24** (Orphan Message). $\langle\, \Delta \,;\, \Theta \,\rangle$ has an *orphan message* if there exists $\mathbf{w} \in \Delta(\mathbf{pq})$ but there exists no transition such that consumes it, i.e. there is no transition $\langle\, \Delta \,;\, \Theta \,\rangle \xrightarrow{\alpha*} \langle\, \Delta' \,;\, \Theta' \,\rangle$ with $\mathbf{pq}?|\mathbf{w}| \in \alpha*$.

▶ **Example 25** (Orphan Message). Orphan messages can occur whenever a send prefix is not coupled with the corresponding receive, thus leaving a message *hanging* in the corresponding buffer. For example, the following situation contains an orphan message:

$$\langle\ [\qquad\qquad]\ ;\ \mathbf{p} :: \mathbf{q}!l[U].\ \mathbf{q}?l[U].\ \mathsf{end}, \mathbf{q} :: \mathbf{p}!l[U].\ \mathsf{end}\ \rangle \xrightarrow{\mathbf{pq}!l}$$
$$\langle\ [\mathbf{qp} :: l[U]]\ ;\ \mathbf{p} :: \mathbf{q}?l[U].\ \mathsf{end}, \mathbf{q} :: \mathbf{p}!l[U].\ \mathsf{end}\ \rangle \xrightarrow{\mathbf{qp}!l\cdot\mathbf{pq}?l} \langle\ [\mathbf{qp} :: l[U]]\ ;\ \mathbf{p} :: \mathsf{end}, \mathbf{q} :: \mathsf{end}\ \rangle$$

At the end of the execution, the configuration contains a non-empty buffer: $\mathbf{qp} :: l[U]$.

Another example of orphan messages is one in which the reduction gets stuck because of receiving a message of the wrong type or label, i.e. there is a reception error.

$$\langle\ [\qquad\qquad]\ ;\ \mathbf{p} :: \mathbf{q}!l[\mathsf{int}].\ \mathsf{end}, \mathbf{q} :: \mathbf{p}?l[\mathsf{bool}].\ \mathsf{end}\ \rangle \xrightarrow{\mathbf{pq}!l}$$
$$\langle\ [\mathbf{qp} :: l[\mathsf{int}]]\ ;\ \mathbf{p} :: \mathsf{end}, \mathbf{q} :: \mathbf{p}?l[\mathsf{bool}].\ \mathsf{end}\ \rangle \overset{\alpha}{\not\to}$$

In this case, reduction cannot continue, and the message $\mathbf{qp} :: l[\mathsf{int}]$ cannot be consumed, because $\mathbf{q}$ is expecting payload type bool.

Proving that DMst guarantees the absence of orphan messages relies on the absence of *blocked local types*. A blocked local type is a local type that contains a nested session that cannot terminate, followed by a non-empty continuation. For example, if $L = \triangleright[\mu \mathbf{t}.\mathbf{q_2}!l'[U'].\ \mathbf{t}].\ \mathbf{p}?l[U]$, then L is blocked, because it will enter the nested protocol (with local type $\mu \mathbf{t}.\mathbf{q_2}!l'[U'].\ \mathbf{t}$), but it will never be able to continue executing $\mathbf{p}?l[U]$.

▶ **Definition 26** (Blocked Participant). A *blocked* local type is one that contains a continuation of the form $\triangleright[L_1].\ L_2$, where: (a) L_1 is blocked, or (b) $L_2 \neq \mathsf{end}$ and end is not reachable from L_1.

▶ **Definition 27** (Liveness). We say that $\langle\ \Delta\ ;\ \Theta\ \rangle$ is *live*, if no participant is stuck. A participant $\mathbf{p}$ is stuck in a configuration whenever it cannot progress, i.e. if $\Theta(\mathbf{p}) = L$ with $L \neq \mathsf{end}$, but there is no trace $\langle\ \Delta\ ;\ \Theta\ \rangle \xrightarrow{\alpha*} \langle\ \Delta'\ ;\ \Theta'\ \rangle$ with $\mathbf{p} = \mathsf{subj}(\alpha)$ and $\alpha \in \alpha*$.

▶ **Example 28** (Stuck Participant). The following configuration is *not live*, because even if $\mathbf{p}$ and $\mathbf{q}$ can continue interacting, $\mathbf{r}$ and $\mathbf{s}$ are stuck in a local receive cycle:

$$\langle\ []\ ;\ \mathbf{p} :: \mu \mathbf{t}.\mathbf{q}!l[U].\ \mathbf{t}, \mathbf{q} :: \mu \mathbf{t}.\mathbf{p}?l[U].\ \mathbf{t}, \mathbf{r} :: \mathbf{s}?l[U].\ L_3, \mathbf{s} :: \mathbf{r}?l[U].\ L_4\ \rangle \xrightarrow{\mathbf{pq}!l\mathbf{qp}?l}$$
$$\langle\ []\ ;\ \mathbf{p} :: \mu \mathbf{t}.\mathbf{q}!l[U].\ \mathbf{t}, \mathbf{q} :: \mu \mathbf{t}.\mathbf{p}?l[U].\ \mathbf{t}, \mathbf{r} :: \mathbf{s}?l[U].\ L_3, \mathbf{s} :: \mathbf{r}?l[U].\ L_4\ \rangle \xrightarrow{\alpha*} \ldots$$

No possible trace can contain $\mathbf{rs}?l$ or $\mathbf{sr}?l$. Participants $\mathbf{r}$ and $\mathbf{s}$ are stuck. Note that, from Definition 19, only receive prefixes can get stuck, since send prefixes will always succeed.

▶ **Theorem 29** (Orphan Message Freedom and Liveness).
Suppose $\langle\ \Delta\ ;\ \Theta\ \rangle \leqslant [\![G]\!]$*, such that* Θ *contains no blocked participants. Then* $\langle\ \Delta\ ;\ \Theta\ \rangle$ *is* free of orphan messages *and* live.

Proof. Liveness is a straightforward consequence of orphan message freedom. The prefix of a local type can have two kinds of actions: outputs (sending data or invitations), or inputs (receiving data, or accepting invitations). Every input is coupled with an output by another participant (see Definition 13). Hence outputs can always be performed, in any state. To prove that inputs can always be consumed, we use trace equivalence. We show that any pending message can be received, since a step can only happen in a continuation if its subject is not in any of the previous prefixes, and it is always possible to end nested protocols, because they cannot be blocked. ◀

▶ **Proposition 30.** *As a consequence of Theorems 23, 29 and 20, the global types of Example 10 are live and deadlock-free.*

4 GoScr Code Generation

This section describes the GoScr toolchain. GoScr is an extension of nuScr [48], which is a new implementation of Scribble in OCaml. nuScr is designed with modularity and extensibility in mind, so that extensions of the core MPST theory [60] can be easily integrated.

GoScr Global Protocols. The syntax of GoScr global protocols is given in Definition 31.

▶ **Definition 31** (GoScr syntax).

$$P ::= \texttt{global protocol}\ x(\texttt{role}\ \mathbf{p}_1, \dots, \texttt{role}\ \mathbf{p}_n; \texttt{new role}\ \mathbf{q}_1, \dots, \texttt{role}\ \mathbf{q}_m)\ \{P^*\ G\}$$

$$g ::= m[U]\ \texttt{from}\ \mathbf{p}\ \texttt{to}\ \mathbf{q} \mid \mathbf{p}\ \texttt{calls}\ x(\mathbf{p}_1, \dots, \mathbf{p}_n)$$

$$G ::= \texttt{choice at}\ \mathbf{p}\ \{G_1\}\ \texttt{or} \dots \texttt{or}\{G_n\} \mid g;\ G \mid \texttt{rec}\ \mathrm{t}\ \{G\} \mid \texttt{continue}\ \mathrm{t} \mid \texttt{end}$$
$$\mid\ \texttt{continue}\ \mathrm{t}\ \texttt{with}\{g_1;\ \dots g_n;\ \mathbf{p}\ \texttt{calls}\ x(\mathbf{p}_1, \dots, \mathbf{p}_n)\} \mid \texttt{do}\ x(\mathbf{p}_1, \dots, \mathbf{p}_n);\ G$$

A GoScr module is a sequence of one or more `global` protocols. The last protocol definition is the entry point. The constructs of GoScr were chosen to mirror those defined in Definition 1: `global protocol` are protocol definitions ($x = \lambda\langle\vec{\mathbf{p}}; \nu\vec{\mathbf{q}}\rangle.G\dots$; `global protocol` can be used for protocol declarations with no new participants; `choice at` $\mathbf{p}$ defines directed choices from $\mathbf{p}$ to the receiver of the first interaction in the G_i; `do` is a protocol call to a global protocol; and the rest of the constructs correspond to those of DMst. Protocol definitions in GoScr can start by defining other nested protocols, but this is simply a syntactic convenience, since we require every role in a nested protocol to be bound by the protocol signature.

Steps for Code Generation. The steps of code generation in GoScr are: (1) lifting all nested protocol definitions to the top-level; (2) obtaining the projections of all roles in all protocol definitions; (3) preprocessing local types to deal with instances of ⧫ (or `continue`...`with`... in GoScr); and, (4) translating the local types to Go functions, where communication is implemented using Go channels, interleaved with *callbacks* that will be used to implement the program logic.

Step (1) is straightforward. Step (2) is an implementation of Definition 13. Step (3) requires applying local type equivalences to unfold any updatable recursion. Step (4) traverses the local types, and generates on demand the necessary channels and callback interfaces. The type of the Go channels is an interface that represents the allowed payload types. Then, for each labelled message exchange: (1) we add a new type declaration for the label and payload type that implements the interface of allowed messages; (2) we search for a channel for the required endpoints, creating it if necessary; (3) we create the necessary callbacks before or after the interaction. The channels can be created either synchronous, or buffered with a user-specified size. Choosing synchronous channels is safe, since the traces accepted by using synchronous semantics is a subset of those accepted by our asynchronous semantics, which implies that the same safety properties will hold.

Go Code Generation. The Go code for each role and protocol is generated in `protocol/`. Communication is implemented using regular Go send/receive statements. There is no need to explicitly send message labels, since labels are encoded as type declarations. Protocol choices are encoded as type switches, either on the value returned from a previous callback (internal choices), or on the received value (external choices). We only generate implementations for branching choices that start with an explicit interaction.

Calling a nested protocol is implemented as regular Go function calls. `rec` constructs are generated as labelled `for` loops, where the body of the recursion is used to generate the body of the `for` loop, and recursive variables are translated as `continue` to the label of the corresponding variable. It is also possible to represent recursion using protocol calls.

```go
func BFib_F2(ctx Ctx_BFib_F2, wg *sync.WaitGroup, ch_F2_F3, ch_F3_F2 chan MsgBFib)  {
    x := ctx.Send_F3_BFib_Fib2()        // Callback to generate payload
    ch_F3_F2 <- x                       // Send payload to F3
    x_1 := <- ch_F2_F3                  // External choice by from F3
    switch v := x_1.(type) {
    case End:                           // F3 chooses to finish the protocol
        ctx.Recv_F3_BFib_End(v)         // Callback for processing label End
        ctx.End()
        return
    case Call_F1_BFib:                  // F3 sends the channel for acting as F1 in BFib
        ctx_1 := ctx.Init_F1_BFib_Ctx() // Initialise context for F1 in BFib
        BFib_F1(ctx_1,wg,v)             // Run code for F1 in BFib with channel [v]
        ctx.End_F1_BFib_Ctx(ctx_1)      // Close context for F1 in BFib
        ctx.End()
        return
    } }
```

Figure 5 Implementation of role F2 in protocol BFib

However, protocol calls would need to create the necessary channels and send them to any participant in the protocol, thus being less efficient than using `rec` and `for`.

4.1 Linearity and CFSM Code Generation

Program logic is defined through *callbacks*, similar to [42, 62], to avoid the *linearity* problem of previous Communicating Finite State Machine approaches (e.g. [3]). In a CFSM approach, code generation from a local protocol produces a series of interfaces that encode the protocol states. Each protocol state exposes *only* the permitted actions (e.g. send/receive), and returns the next state in the protocol. Programmers must use such states to implement their program logic. The linearity problem arises from the fact that nothing prevents programmers from mistakenly using the same protocol state again. For example, suppose that `st0`, `st1`, . . . , are protocol states that expose different send/recv actions. A programmer might (mistakenly) save state `st1` and perform its action twice in the implementation. In the Go code snippet below, `st1` is used both in Line 2 and Line 4, violating linearity:

```go
1  st1 := st0.send_Msg_to_p(x)
2  st2 := st1.recv_Lbl_from_p(&z)
3  ...
4  stn := st1.recv_Lbl_from_p(&buffer) /* linearity error at st1 */
```

If participant `p` does not send any other message, then this implementation will **deadlock**. If `p` does send another message, this might cause a run-time error. A callback-based approach solves this problem *by construction*, since channels are not exposed to programmers [42, 62].

4.2 Example of Generated Go Code

Consider the following GoScr global type:

```
global protocol BFib(role Res, role F1, role F2; new role F3) {
  Fib1(v:int) from F1 to F3; Fib2(v:int) from F2 to F3;
  choice at F3 {
    F3 calls BFib(Res, F2, F3);
  } or {
    Result(fib:int) from F3 to Res; End() from F3 to F2; }}
```

This is a *bounded* version of Example 4, that computes the Fibonacci sequence up to an upper bound. `F1` and `F2` send their respective $n-2$ and $n-1$ Fibonacci numbers to `F3`. Then, `F3` computes the n-th number, and makes a choice: compute the $n+1$ number, or end the protocol. If `F3` decides to continue, then a recursive call to `BFib` happens. Otherwise, it sends the result to Res, and notifies F2 that the protocol is ending. F3 needs to notify F2, because depending on F3's decision, F2 may needs to forward its $n-1$ number.

Figure 5 shows the code for F2 in BFib. The parameters of BFib_F2 are: ctx is the local state for F2; wg is used to ensure that the main thread does not resume execution until all participants have finished executing; ch_A_B is the channel for communicating from B to A. The first interaction of F2 is a message to F3. The payload for this message is generated in Line 2, it is sent in Line 3. Then, F3 makes a choice: either it sends the result back to Res and sends End to F2 to communicate the end of the protocol, or it calls BFib recursively. F2 performs a type switch to check which branch it needs to take (Line 4). If the label it receives is End (Line 6), then F2 processes this label and ends the protocol. Otherwise, F2 receives an invitation as F1 in BFib (Line 10); then F2 initialises a new context for F1 using the callback on Line 11; it calls BFib_F1 with this new context, the waitgroup, and the received channel (Line 12); F2 performs cleanup on the context for F1, gathering any necessary results from the call (Line 13); and, finally F2 finishes.

Finally, GoScr generates the main protocol entrypoint, which creates the goroutines for F1, F2 and F3, all the needed channels, and waits for the completion of the protocol.

Usability and GoScr Front-end. The tool requires the user to instantiate a large number of callbacks and interfaces to allow running a protocol. Since the GoScr methodology is top-down, the user must start by specifying a protocol. Therefore we expect an end-user to be aware of the callbacks and contexts that need to be instantiated. However, many of such instantiations are tedious, but straightforward, and can be automated in future work. We discuss this improvement in Section 7.

Deadlock Freedom and Liveness. Since the generated code follows the behaviour of the local types, it will satisfy both **deadlock freedom** and **liveness** (Theorems 23 and 29). Although the generated code satisfies these properties, whether the final code that is run also satisfies them depends on three requirements on the callbacks. These requirements are not checked by GoScr, and must be guaranteed by GoScr users. The three requirements that the callbacks must satisfy are: (1) callbacks must not have side-effects that interfere with other participants (e.g. using channels to add communication that is not accounted for in the protocol) (2) callbacks must be terminating, otherwise a participant may block before a necessary interaction, in a non-terminating callback; and, (3) callbacks must ensure that nested protocol calls that are not in tail position are terminating. Requirement **(1)** is to guarantee that programmers do not use local synchronisation mechanisms that are not accounted for in the protocol, and can cause blocking. Requirement **(3)** is to guarantee that any interaction after a nested protocol call is eventually performed. GoScr checks that local types are not blocked (Definition 26), so the code for nested calls that are not in tail position will always contain a path that ends the protocol. However, whether the actual code is terminating depends on the callback implementation that the users need to provide satisfying Requirement **(3)**. Provided that these requirements are met, and assuming a fair scheduler, GoScr implementations will be deadlock free and live by construction. These requirements are not unique to our implementation. Similar requirements must be satisfied in other MPST code generation approaches.

5 Evaluation

We evaluate three aspects of GoScr: (1) the runtime overhead of the GoScr backend (§ 5.1); (2) the increased expressiveness with respect to related approaches (§ 5.3); and (3) the applicability of GoScr for building realistic protocols, by implementing dynamic task del-

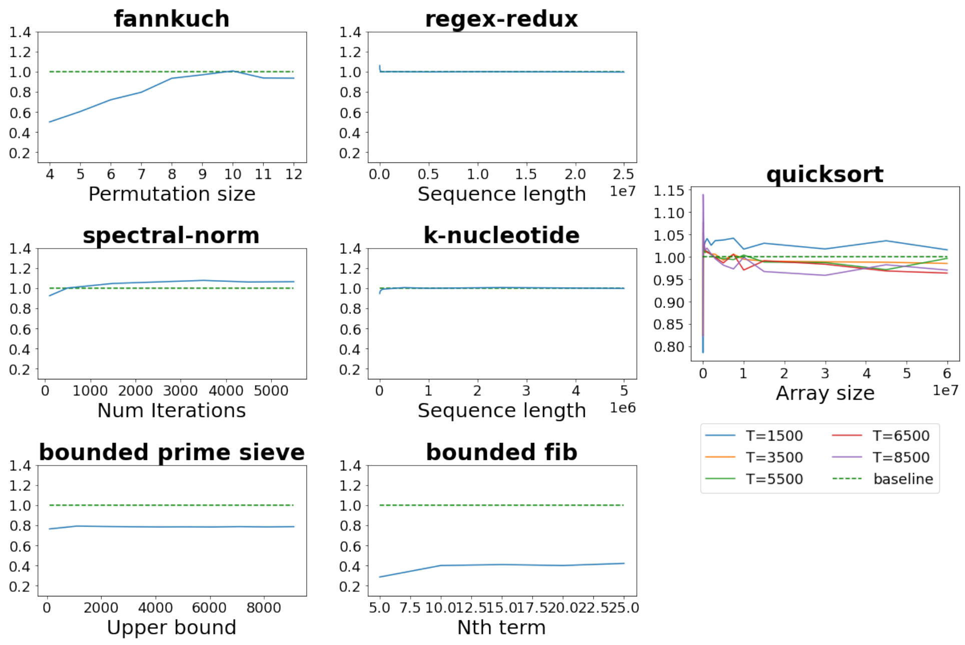


Figure 6 Execution time comparison (t_{base} / t_{GoScr}), CLBG and Quicksort.

egation, a Domain Name System, and a parallel Min-Max strategy. We show that for computation-intensive protocols, the runtime overhead of GoScr is negligible.

5.1 Runtime Overhead of GoScr

We use an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processor with 4 physical cores, 16GB RAM, running Ubuntu 16.04.7 and Go version `go1.15.11`. We use Golang's time package to measure execution times. There are two main sources of run-time overheads: (1) callbacks; and (2) type switches and assertions. Our approach is to compare GoScr implementations against baseline Go code. Baselines are taken from benchmarking repositories, and follow similar communication patterns to the GoScr implementations. The measured time includes session initialisation. We execute each benchmark for a minimum of 20 iterations and a minimum of 20 seconds. The standard deviation for computationally expensive benchmarks is less than 5%. Only the standard deviation of fibonacci and prime sieve with small inputs ($<$ 10th term, bound $<$ 2000) remain high, at 70%. This is because these benchmarks with very short execution times (in the order of nanoseconds) are highly dependent on the system (e.g. channel creation, goroutine scheduling, etc). Our benchmarks are mainly taken from the *Computer Language Benchmarks Game* [17], and we include a parallel Quicksort that showcases the handling unbalanced workloads. Figure 6 shows the execution time of the Go baseline relative to GoScr: t_{base} / t_{GoScr} (below $y = 1$ is a slowdown, above is a speedup).

Computer Language Benchmarks Game (CLBG). CLBG [17] is a repository of programs used to compare the performance of different languages. We use four concurrent Go programs: (1)`fannkuch` counts the maximum number of flips for a permutation of length n; (2)`regex` matches regex patterns in a DNA string; (3)`spect` (spectral-norm) calculates the greatest eigenvalue of a matrix; and (4)`k-nuc` (k-nucleotide) counts the occurrences of a molecule sequence in a DNA string. We selected these benchmarks out of [17] because they parallelise the work using goroutines and channels, following a similar scatter/gather approach that depends on runtime values, and they could not be accurately captured by previous MPST approaches. We use the CLBG implementations [17] as the Go baseline implementations,

```
global protocol DynTaskGen(role S;
    new role W) {
  choice at S {
    Req(req: string) from S to W;
    S calls DynTaskGen(S);
    choice at W {
      Resp(resp: string) from W to S;
    } or {
      Error(err: string) from W to S; }
  } or {
    LastReq(req: string) from S to W;
    choice at W {
      Resp(resp: string) from W to S;
    } or {
      Error(err: string) from W to S;
    }}}
```

```
global protocol ClientServer(role C,
    role S) {
  rec REPEAT {
    Req(req: string) from C to S;
    S calls DynTaskGen(S);
    choice at S {
      Resp(resp: string) from S to C;
      continue REPEAT;
    } or  {
      Error(err: string) from S to C;
      continue REPEAT;
    }}}
```

Figure 7 GoScr protocol for Dynamic Task Generation

and we extracted the communication structure of the baseline implementations as GoScr protocols. A single execution for each of these protocols takes between 1 millisecond–10 seconds depending on the input size. Smaller input sizes imply smaller local computation times, and therefore, the overhead introduced by GoScr will be more significant. We can observe a slowdown of up to 50%, in `fannkuch`, for executions in the order of magnitude of milliseconds. However, as the workload increases, the difference in the execution time shrinks to the point of becoming negligible, as we can observe in Figure 6. The `regex` baseline has a high standard deviation, which explains the small peak for the first result of `regex`, since when the execution time is in the order of hundreds of microseconds, the non-deterministic scheduling of the goroutines can significantly affect the results. `spect` seems to show that for large enough values, the GoScr implementation performs better than its naively implemented counterpart. However, the real difference in the execution time is negligible, and it is explained by differences in the program structure, e.g. the baseline uses a single shared channel, whereas GoScr generates different channels for every new goroutine.

Microbenchmarks. Bounded fibonacci (`fibonacci`) shows, as expected, that the overhead of performing type switches and callbacks is relatively high when compared with a simple addition. The baseline runs in in 40% of the execution time of GoScr. Bounded prime sieve (`prime`) shows that, when the computation complexity increases slightly (modulus operation on a stream of values), then the GoScr version performs in about 80% the execution time of the baseline. In both cases, when we add more participants and interactions to the protocol (larger values on the x-axis) the overhead *remains constant, and does not increase.*

Unbalanced Workload. In *Parallel QuickSort* (`qsort`), workers either partition the array and spawn two new workers, or apply a sequential Quicksort, depending on a threshold size (T). The execution times are similar to the CLBG benchmarks (50 microseconds–2 seconds). We observe a negligible difference in the execution time for different threshold sizes, and a spike for small arrays due to the high standard deviation for array sizes under the threshold. GoScr execution times are in the range of 1.05 and 0.95 times the baseline.

5.2 Use Cases

We demonstrate the expressiveness of GoScr using three applications, all of which require dynamic participants, and could not be expressed by previous work [3, 37].

(a) **Dynamic Task Generation**: We present a correct implementation of the program in Figure 1 using GoScr. It is a master-worker pattern with dynamic participants.

(b) **Domain Name System (DNS) protocol**: We demonstrate how GoScr can be used to

Protocol	Dyn	Unb	Inv	DMst	[3]	[37]	[7]	[9]
1. `Dynamic Ring`	●		●	✓	✗	✗	✓	✗
2. `Dynamic Pipeline`	●		●	✓	✗	✗	✓	✗
3. `Dynamic Recursive Pipeline`	●	●		✓	✗	✗	✗	✗
4. `Dynamic Recursive Tree`	●	●		✓	✗	✗	✗	✗
5. `Dynamic Recursive Task Gen.`	●	●		✓	✗	✗	✗	✗
6. `Dynamic Fork-Join`	●			✓	✗	✗	✓	✗
7. `Recursive Fork-Join`	●			✓	✗	✗	✓	✗
8. `Bounded Fibonacci` [3]	○		○	✓	△	△	✓	✗
9. `Unbounded Fibonacci`	●	●		✓	✗	✗	✓	✗
10. `Fannkuch-Redux` [17]	○		○	✓	△	△	✓	✓
11. `Spectral-Norm` [17]	○			✓	△	△	✓	✓
12. `Regex-Redux` [17]	○			✓	✓	✓	✓	✓
13. `K-Nucleotide` [17]	○			✓	✓	✓	✓	✓
14. `Bounded Prime Sieve`	●			✓	✗	✗	✓	✗
15. `Dynamic Task Generation`	●	●		✓	✗	✗	✓	✗
16. `Domain Name System` [28]	●		●	✓	✗	✗	✓	✗
17. `Noughts and Crosses` [42, 49]	●		●	✓	✗	✗	✓	✗

Dyn: Dynamic participants; Unb: Unbounded participants; Inv: Choice through invitations

Table 1 Comparison of Expressiveness

specify one of the core Internet protocols, modelling as dynamic participants the different DNS servers which may need to be contacted in order to resolve a host's IP address.

(c) **Noughts and Crosses with Min-Max** [49]: We implement a Min-Max strategy for the well-known two-player game of Noughts and Crosses to demonstrate the suitability of DMst to model a parallel Divide and Conquer paradigm.

Dynamic Task Generation. The aim of this program is to generate the first n square numbers by delegating the calculation of each square number to a different worker goroutine. The program uses a common computation in Go, the *master-worker pattern*, where goroutines dynamically divide and delegate part of their tasks to other goroutines, aggregating their partial results to produce the complete result. We highlighted in § 1 (Figure 1) how even in such a simple example, incorrect management of channels can lead to orphan messages and deadlocks. Figure 7 shows a GoScr protocol specification whose behaviour is a safe version of the program in Figure 1. Notice how the behaviour of the `select` statement in Figure 1 is represented as a choice. In Figure 7, the `ClientServer` protocol models the behaviour of the main loop of the program, where two roles, a client and a server, repeatedly exchange requests (Line 14) and responses (Line 17). The server may also communicate an error in the computation of the request to the client (Line 20). We model the master-worker pattern as a call to protocol `DynTaskGen` (Line 15). Every call to the protocol introduces a new worker (`W`), and the master (`S`) will delegate a task to each new worker (Lines 4,11). If there are are more tasks to assign, it will assign those tasks to new workers through recursive calls to `DynTaskGen` (Line 5). Once it has assigned the final task (Line 11), it will traverse the protocol stack, aggregating the results from the different workers in reverse order (Lines 7,13). While computing their subtask, the workers may encounter an error which they will communicate back to the server (Lines 9, 15). As opposed to the original program in Figure 1, the server will continue aggregating all the results from the workers even after encountering an error in order to ensure that there are no orphan messages.

5.3 Expressiveness

We compare the expressiveness of GoScr against the parameterised Scribble [3] and the static analysis framework of Go [37]. For a reference purpose, we also list comparisons with **theory-only work** in [7, 9] (i.e., they are ***not*** implemented). See § 6 for more detailed comparison with [7, 9]. In Table 1, we present the protocols that we implemented and whether or how closely other approaches [37, 3] can represent them. All our DMst-based implementations introduce dynamic, possibly unbounded participants. All representable protocols (✓) by DMst in Table 1 are deadlock-free and live. For protocols which can be modified and re-implemented with [37] or [3], we use ○. Protocol 3 cannot be captured by any of the previous work, since it requires the dynamic introduction of participants to a recursive protocol. [3, 37] could only precisely model Protocols 12 and 13, as they create all the participants at the start. In Protocols 8, 10 and 11, the goroutines are spawned and assigned tasks dynamically, but [3, 37] can model them by initialising all goroutines at the start. We write △ to represent such changes to protocol structure. Three use cases (Protocols 13–15) discussed in § 5.2, could not be expressed by [3, 37]. In summary, DMst is more expressive than [3, 37], and capture more closely the typical Go programming style.

6 Related Work

There are a vast amount of studies of session types [27, 15, 1]. Due to the space limitations, we only compare with the most closely related work on multiparty session types (MPST).

Binary Session Types. While Scalas et al. [50] prove that the MPST processes can be mimicked by linearly typed processes with a continuation-passing style translation, in general, it is not possible to guarantee deadlock-freedom for more than two interleaved binary session processes unless one uses additional sophisticated means such as a global causal analysis on channels (e.g. [12, 4, 5]), graph-connectivity analysis with extensions on fork primitives [29], and event-driven constructs [57, 24, 34]. GV, a linear functional language with binary session types, can guarantee deadlock freedom by relying on linear typing [58]. However, linear typing prevents cyclic topologies that change dynamically, since this would require a participant to drop their communication channels when new participants join, as in Example 7. There are further substantial differences with our approach. First, GV is an end-point calculus, whereas DMst's global types are *global specifications*, from which we can extract endpoint Go code (GoScr). Secondly, while both GV and DMst support similar programming patterns (e.g. pipeline and tree-like topologies, and channel passing), there are two major differences. Both GV and GoScr support sending effectful functions over channels (e.g. using `chan func()` type in Go, and passing a generated protocol implementation), GV's type system would guarantee deadlock-freedom, but in Go, it would depend on how the function is used (requirements 1-3 in Section 4).

Code Generation and Multiparty Session Types. We follow the standard MPST top-down *specification-guided* methodology to guarantee **safety and liveness properties by construction using code generation**, extending an extensible toolchain, nuScr [48]. Safety by construction via code generation is a common approach in MPST. Scribble is a language/tool [51, 48] used for generating APIs for safely implementing distributed systems written in the end-point programming language that are guaranteed to conform to a protocol, and are therefore deadlock-free [25]. This approach has been applied to several languages, e.g. Scala [50, 57], Java [33], F# [43], Go [3], TypeScript [42], F⋆ [62] and Rust [35, 6]. A later extension of [25] proposed *explicit connection actions* as part of the Scribble protocol [26], which is also recently applied to domain-specific language in [19]. This construction specifies the point in the protocol where the different participants join, but the role of these participants must be statically known. Hence it does not allow the unbounded participants to change the protocol topology, as DMst does. *Parameterised multiparty session types* extend MPST with a parametric number of participants [11]. One example is the work by Castro-Perez et al. [3], discussed it in § 1. *Pabble* [46, 45] is another parameterised extension of Scribble used for generating safe by construction C+MPI code. Zhou et al. [62] formalised and implemented an extension of MPST with *refinement types*, which can specify constraints in the messages. Their backend targets F⋆, and follows a similar callback approach to the one in this paper. Miu et al. [42] define an extension of MPST for web programming in TypeScript that uses the callback approach. Unlike DMst, the participants in all these approaches are fixed from the start of the protocol. Viering et al. [57] present a theory and implementation of MPST aimed at programming correct fault-tolerant distributed systems that supports the dynamic replacement of participants in a protocol. In their work, the replacement of participants must happen within some known roles, and their global types do not allow to extend the current protocol interactions with those of new participants. Viering et al. [57] use event handlers in their code generation, which allows safe session interleaving. Instead, we use an operator to *combine* global types in a way that does not introduce deadlocks. All previous work, unlike DMst, does not support dynamically growing protocols with an unbounded number of participants such as Example 6. Jacobs et al. [30] extend GV, a binary session typed calculus with multiparty session types. The calculus allows the introduction of new participants, but the protocols themselves are restricted to a fixed set of participants. Their use of linearity prevents the definition of recursive dynamic topologies, unlike DMst.

Dynamic Multiparty Session Types. Dynamic multirole session types (MRST) enable a set of participants which belong to the same group (i.e. role) to join a multiparty session type [9]. The major limitations are: (a) all the roles are fixed at the start (b) participants can only join at specific points in the protocol: (1) at the beginning of each iteration of a recursive protocol; or (2) at particular points marked with explicit barriers and locks. We list a number of protocols that cannot be represented using MRST in Table 1. In contrast, DMst allows any arbitrary role to join at any nested session call. A nested session call is a form of delegation, which is not supported by MRST. Therefore, a protocol such as a dynamically growing pipeline (e.g. Fibonacci in Example 4) cannot be represented by [9] either, since it would require participants to evolve their behaviour through channel passing. Nested multiparty session types [7] allow multiparty protocols with unbounded, dynamic participants. However, [7] cannot represent *recursive* protocols that are *updated* with new dynamic participants. Hence the main example of this paper, Example 6, is not representable in [7]. Moreover, nested multiparty session types cannot prove liveness (our Theorem 29), except for non-recursive protocols. Arbitrary session interleaving in [7] can introduce orphan messages. DMst has proven deadlock-freedom and liveness clearly identifying the conditions (Definition 26). This limitation is stated in [7, Proposition 3], i.e. a protocol that violates liveness will be accepted in [7], but not in DMst. Additionally, the theory of DMst has a number of differences that make it better suited for implementing than nested MPST: (a) DMst's choices are more flexible than those in nested MPST, since DMst can also depend on protocol calls; (b) the semantics of nested MPST is synchronous, while DMst is asynchronous; (c) nested MPST does not prove trace equivalence between global types and local type configurations; (d) The syntax of DMst's global types are simpler than those in nested MPST, but more expressive – this is because in nested MPST, protocol definitions are part of the global type syntax, which requires the use of a kinding relation for checking well-formedness. Nested MPST protocols do not allow the occurrence of free roles, and are therefore equivalent to DMst's global types with just top-level protocol definitions, which avoids the kinding relation for checking well-formedness. Due to our simpler but more expressive treatment, DMst is more suitable for real language implementations.

Verification of Go Programs. Our work aims at providing *correctness by construction.* The comparison with the previous code generation approach in Go [3] can be found in **(C)** in § 1 and ***Expressiveness*** in § 5.3. All of the previous work is limited to *bounded* participants.

The following are several recent lines of work on *a posteriori* verification of message passing in existing Go programs. All of them use *whole*-program techniques, and support only the built-in Go channel *primitives* (i.e., intra-process messaging); none of them, however, support a dynamic, unbounded number of participants. Gobra is an automated tool for the modular verification of Go programs, based on separation logic [59]. Gobra is aimed at the functional verification of Go programs, whereas our approach focuses on communication safety. GoScr is fully automated, and aimed at building live and deadlock-free communicating systems by construction. In contrast, Gobra is aimed at the verification of annotated Go code, and it requires a high amount of invariant annotations.

Ng and Yoshida [47] extract graph-based protocol specifications [38] from Go programs that are checked for deadlock-freedom; Stadtmüller et al. [53] extract regex-based protocol specifications [55], 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. [36, 37] (already compared in **(B)** in § 1 and ***Expressiveness*** in § 5.3) statically infer channel communication patterns from Go programs as *behavioural types*, that are checked for liveness properties. This was recently extended to analyse shared memory concurrency [14]. Like previous work, their tool is also limited to verify finite controlled programs, it 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. Liu et al. [39] present a tool that detects blocking misuse-of-channel bugs in Go and produces bug fixes for Go programs. Unlike DMst, Liu et al. [39] focuses only on practice, and does not formalise nor guarantee communication safety, deadlock-freedom nor liveness. Moreover their tool produces both false positive and false negative errors.

7 Conclusion and Future Work

GoScr is the first implementation of multiparty session types with dynamic, unbounded participants, from which we generate Go code with unbounded participants that is, *by construction*, deadlock-free and live. GoScr focuses on correctness (Theorems 20, 23 and 29), and it is strictly more expressive than previous Go verification frameworks (see § 1, Table 1, § 6). Furthermore, we observe that whenever the computation time is large with respect to the communication time, the performance overhead becomes negligible. GoScr is therefore suitable for implementing systems where correctness is prioritised, or systems where the computation times dominate over communication. Currently, DMst does not allow a participant to communicate with an unbounded number of participants during protocol execution. This is a limitation of the Go code generation, which we plan to address in future work. We are also considering extending our back-end to use event-handlers in the style of Viering et al. [57], and allow the arbitrary parallel composition of global types instead of our combination operator. We are also planning to extend the back-end to disparate transports (e.g. using TCP instead of Go channels), thus allowing the implementation of distributed systems. The main challenge of this is integrating delegation, as it is required by protocol invitations, in these disparate transports. Finally, to simplify usability, we plan to extend the protocol specification with annotations to guide code generation, so we can automatically generate trivial callback/context instantiation. We plan to draw inspiration for such annotations from *choreographies*, e.g [31].

References

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

2 Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008 - Concurrency Theory*, pages 418–433. Springer, 2008.

3 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go. In *POPL'19*. ACM, 2019. doi:10.1145/3290342.

4 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *15th International Conference on Coordination Models and Languages*, volume 7890 of *LNCS*, pages 45–59. Springer, 2013.

5 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 26:238–302, 2015.

6 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693. ACM, 2022.

7 Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR 2012 – Concurrency Theory*, pages 272–286. Springer, 2012.

8 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods in System Design*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

9 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL'11*, pages 435–446. ACM, 2011.

10 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP 2013*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013. doi:10.1007/978-3-642-39212-2_18.

11 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

12 Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC 2007*, volume 4912 of *LNCS*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.

13 Docker: Empowering app development for developers. https://www.docker.com/, November 2020.

14 Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs. In *ECOOP'20*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

15 Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. River Publishers series in automation, control and robotics. River Publishers, June 2017.

16 Githut 2.0: A small place to discover languages in github. https://madnight.github.io/githut/#/pull_requests/2020/3, 2020.

17 Issac Gouy. Computer language benchmark game. http://benchmarksgame.alioth.debian.org, 2017.

18 gRPC - a high-performance, open source universal rpc framework. https://grpc.io/, November 2020.

19 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *ECOOP 2021*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.10`.

20 C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., 1985.

21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of 35th Symp. on Princ. of Prog. Lang.*, POPL '08, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

23 Raymond Hu. Distributed programming using Java APIs generated from Session Types. *Behavioural Types: from Theory to Tools*, pages 287–308, 2017.

24 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.

25 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE 2016*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016. `doi:10.1007/978-3-662-49665-7_24`.

26 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE 2017*, volume 10202 of *LNCS*, pages 116–133. Springer, 2017. `doi:10.1007/978-3-662-54494-5_7`.

27 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. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. `doi:10.1145/2873052`.

28 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In *ECOOP 2020*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl, 2020. `doi:10.4230/LIPIcs.ECOOP.2020.9`.

29 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. `doi:10.1145/3498662`.

30 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. `doi:10.1145/3547638`.

31 Sung-Shik Jongmans and Petra van den Bos. A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming. In *ESOP 2022*, volume 13240 of *LNCS*, pages 520–547. Springer, 2022. `doi:10.1007/978-3-030-99336-8\_19`.

32 Kubernetes: Production-grade container orchestration. `https://kubernetes.io/`, June 2017.

33 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, 2016. `doi:10.1145/2967973.2968595`.

34 Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. On Asynchronous Eventful Session Semantics. *MSCS*, 29:1–62, 2015.

35 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages*, volume 12134, pages 127–136. Springer, 2020. `doi:10.1007/978-3-030-50029-08`.

36 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL 2017*, pages 748–761. ACM, 2017.

37 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*, pages 1137–1148. ACM, 2018.

38 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232. ACM, 2015. `doi:10.1145/2676726.2676964`.

39 Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in Go software systems. In *ASPLOS '21*, April 2021.

40 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992. doi:https://doi.org/10.1016/0890-5401(92)90008-4.

41 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41 – 77, 1992. doi:https://doi.org/10.1016/0890-5401(92)90009-5.

42 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in TypeScript with Routed Multiparty Session Types. In *CC 2021*, 2021.

43 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time API generation of distributed protocols with refinements in F#. In *CC 2018*, page 128–138. ACM, 2018. doi:10.1145/3178372.3179495.

44 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. *Models, Languages, and Tools for Concurrent and Distributed Programming*, 11665:236–259, 2019. doi:10.1007/978-3-030-21485-214.

45 Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015. doi:10.1007/978-3-662-46663-6_11.

46 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015. doi:10.1007/s11761-014-0172-8.

47 Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC 2016*, pages 174–184. ACM, 2016. doi:10.1145/2892208.2892232.

48 The nuScr authors. nuscr homepage. https://nuscr.github.io/nuscr/, 2019.

49 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Pearson Education, 2016.

50 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP 2017*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

51 Scribble Authors. Scribble: Describing multiparty protocols. http://www.scribble.org/, 2015. Accessed in Nov. 2020.

52 Stack overflow developer survey 2020. https://insights.stackoverflow.com/survey/2020, 2020.

53 Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static trace-based deadlock analysis for synchronous mini-go. In *APLAS 2016*, volume 10017 of *LNCS*, pages 116–136, 2016. doi:10.1007/978-3-319-47958-3_7.

54 I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003. doi:10.1109/TNET.2002.808407.

55 Martin Sulzmann and Peter Thiemann. Forkable regular expressions. In *LATA 2016*, volume 9618 of *LNCS*, pages 194–206. Springer, 2016. doi:10.1007/978-3-319-30000-9_15.

56 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *ASPLOS 2019*, page 865–878. ACM, 2019. doi:10.1145/3297858.3304069.

57 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *PACMPL*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.

58 Philip Wadler. Propositions as Sessions. In *ICFP'12*, page 273–286. ACM, 2012.

59 Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In *CAV*, pages 367–379. Springer, 2021.

60 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to Multiparty Session Types. In *16th International Conference on Distributed Computing and Internet Technology*, volume 11969 of *LNCS*, pages 73–93. Springer, 2020.

61 T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue. GoBench: A benchmark suite of real-world Go concurrency bugs. In *CGO 2021*. ACM/IEEE, 2021.

62 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *PACMPL*, 4(OOPSLA), 2020.