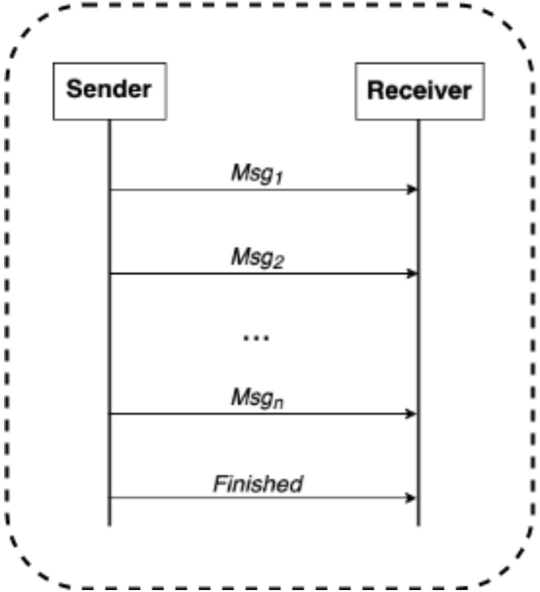
Generating Deadlock-Free and Live Go Code From Unbounded Multiparty Session Protocols

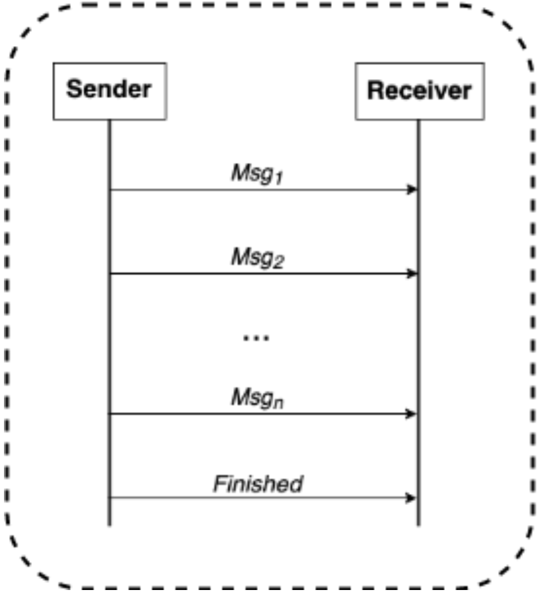
Authors:

David Castro-Pérez, Benito Echarren-Serrano, Nobuko Yoshida

Multiparty Session Types (MPST)



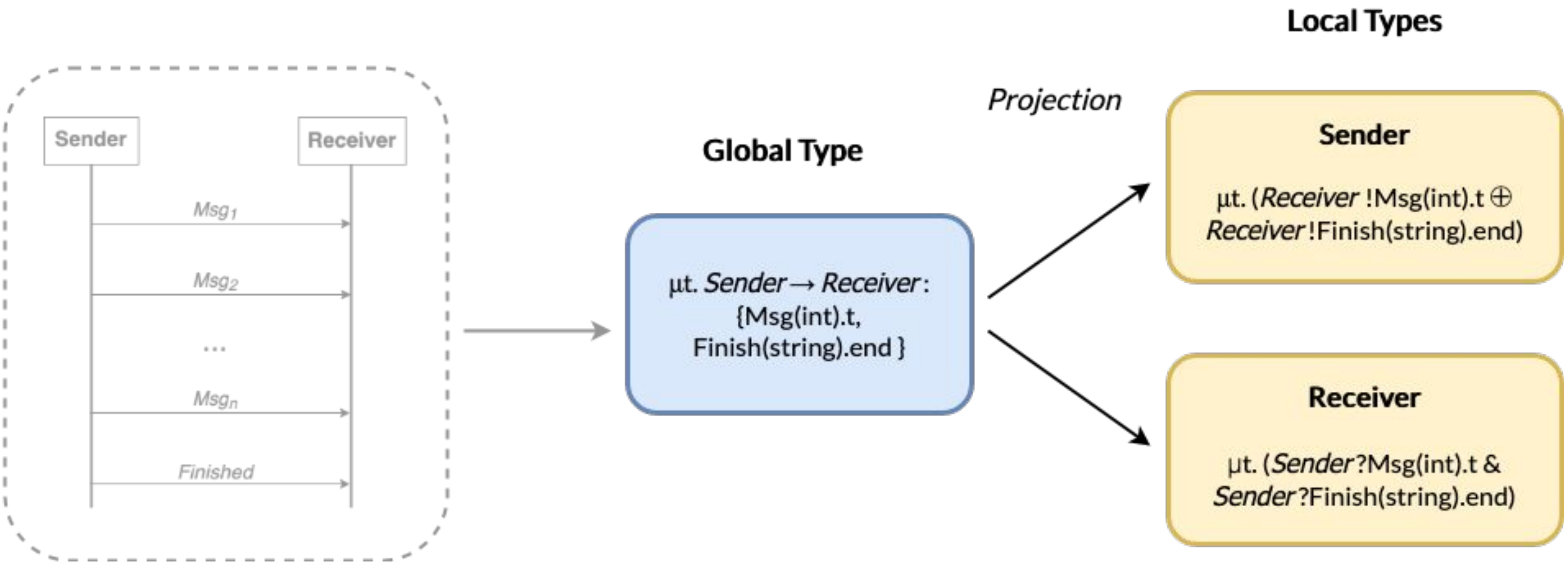
Multiparty Session Types (MPST)



Global Type

$$\mu t.\ \mathit{Sender} \to \mathit{Receiver}: \{\mathrm{Msg(int)}.t,\ \mathrm{Finish(string)}.\mathrm{end}\}$$

Multiparty Session Types (MPST)



Global Type

$\mu t.\ \mathit{Sender} \to \mathit{Receiver}: \{\mathrm{Msg(int)}.t,\ \mathrm{Finish(string)}.\mathrm{end}\}$

Projection

Local Types

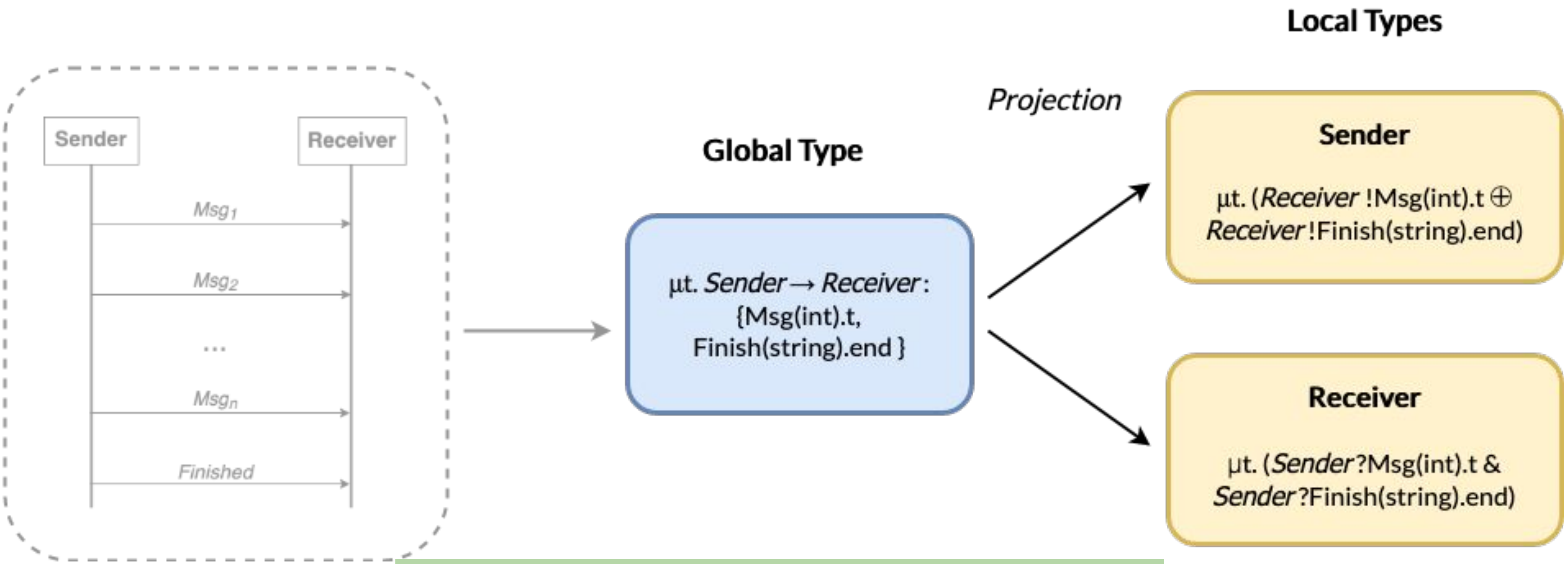
Sender

$\mu t.\ (\mathit{Receiver}\,!\mathrm{Msg(int)}.t \oplus \mathit{Receiver}\,!\mathrm{Finish(string)}.\mathrm{end})$

Receiver

$\mu t.\ (\mathit{Sender}\,?\mathrm{Msg(int)}.t\ \&\ \mathit{Sender}\,?\mathrm{Finish(string)}.\mathrm{end})$

Multiparty Session Types (MPST)



Local Types

Projection

Global Type

$\mu t.\ \mathit{Sender} \rightarrow \mathit{Receiver}:$
$\{\mathrm{Msg(int)}.t,$
$\mathrm{Finish(string)}.\mathrm{end}\}$

Sender

$\mu t.\ (\mathit{Receiver}\,!\mathrm{Msg(int)}.t \oplus$
$\mathit{Receiver}\,!\mathrm{Finish(string)}.\mathrm{end})$

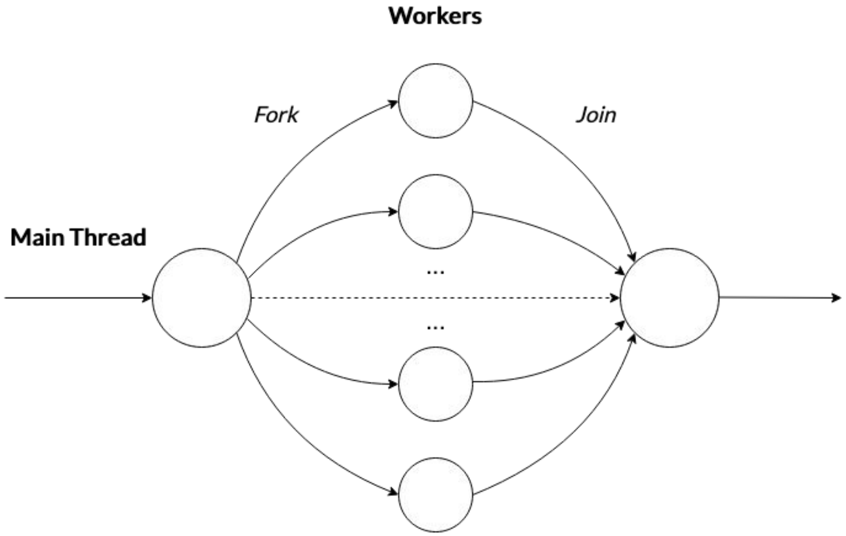
Receiver

$\mu t.\ (\mathit{Sender}\,?\mathrm{Msg(int)}.t\ \&$
$\mathit{Sender}\,?\mathrm{Finish(string)}.\mathrm{end})$

well-typed ⇒ protocol compliance ∧ deadlock freedom

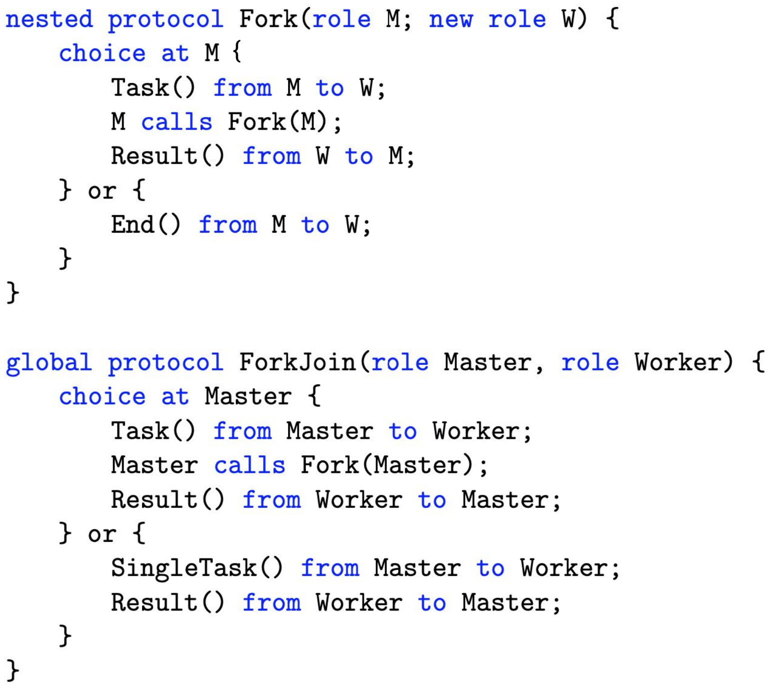
Problems

- In MPSTs, the number of participants **fixed** at the beginning of a session
  - New participants cannot be introduced
- This information may **not available** in many practical settings
- Cannot express common parallel computation patterns
  - **Fork-join** in **Go**



Solution

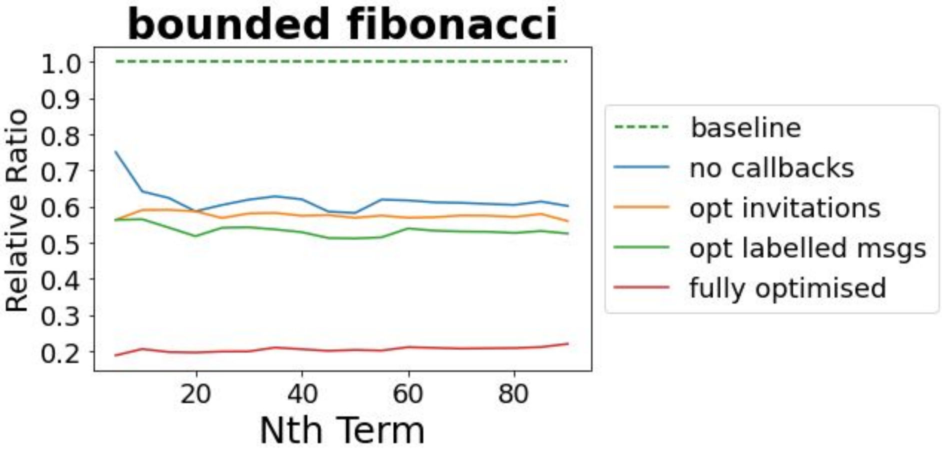
- Based on **unbounded multiparty session types** - an extension to MPST theory:
    - UMP allows protocols to **call** other protocols
    - Participants can be **invited** in protocol calls
    - Protocol calls can involve **new dynamic participants**

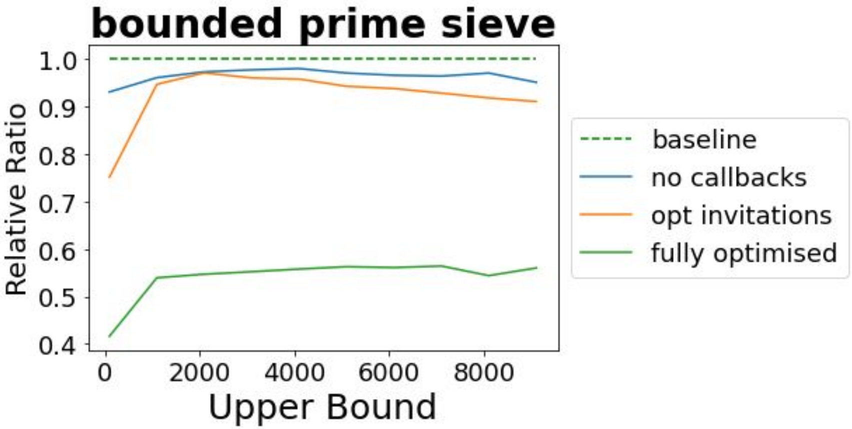


```
nested protocol Fork(role M; new role W) {
    choice at M {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        SingleTask() from Master to Worker;
        Result() from Worker to Master;
    }
}
```

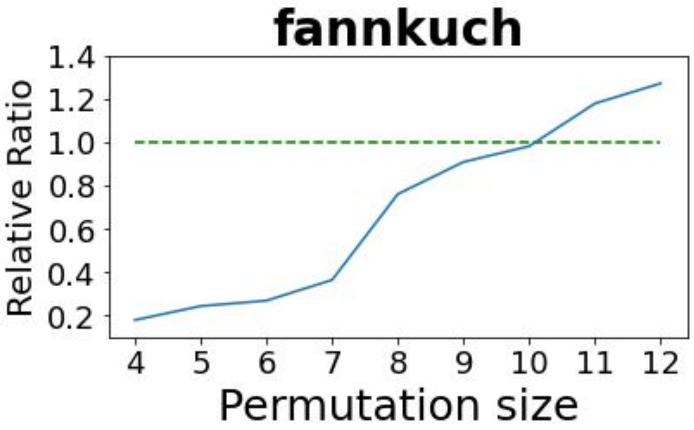
Evaluation

Sources of Overhead

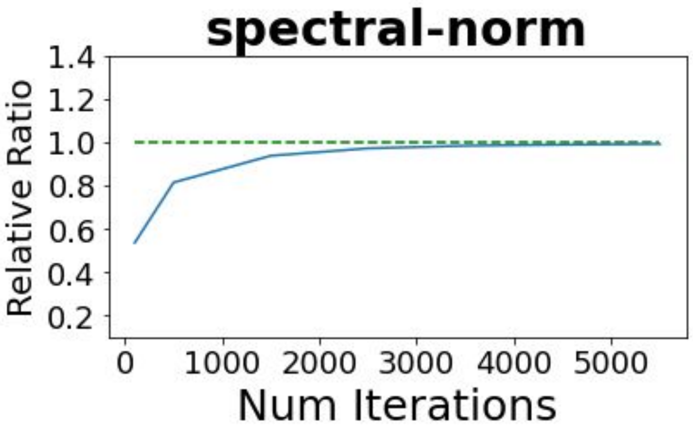




Benchmarks

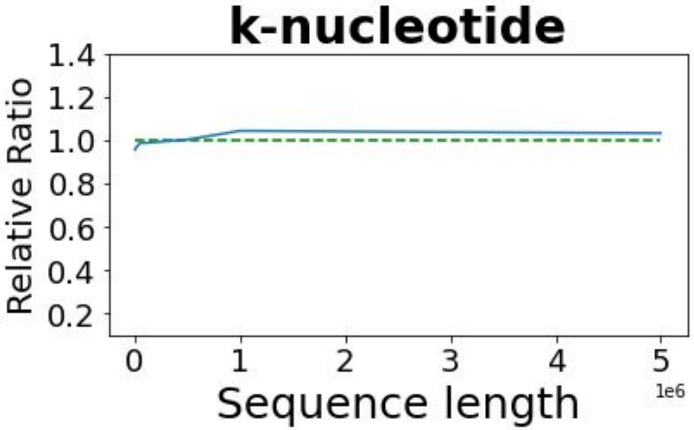


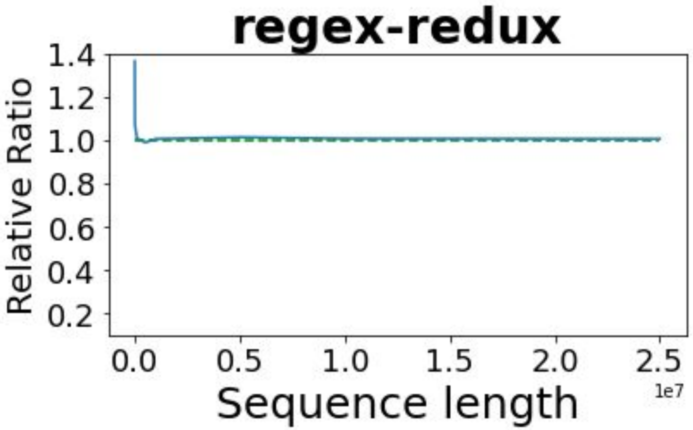
fannkuch



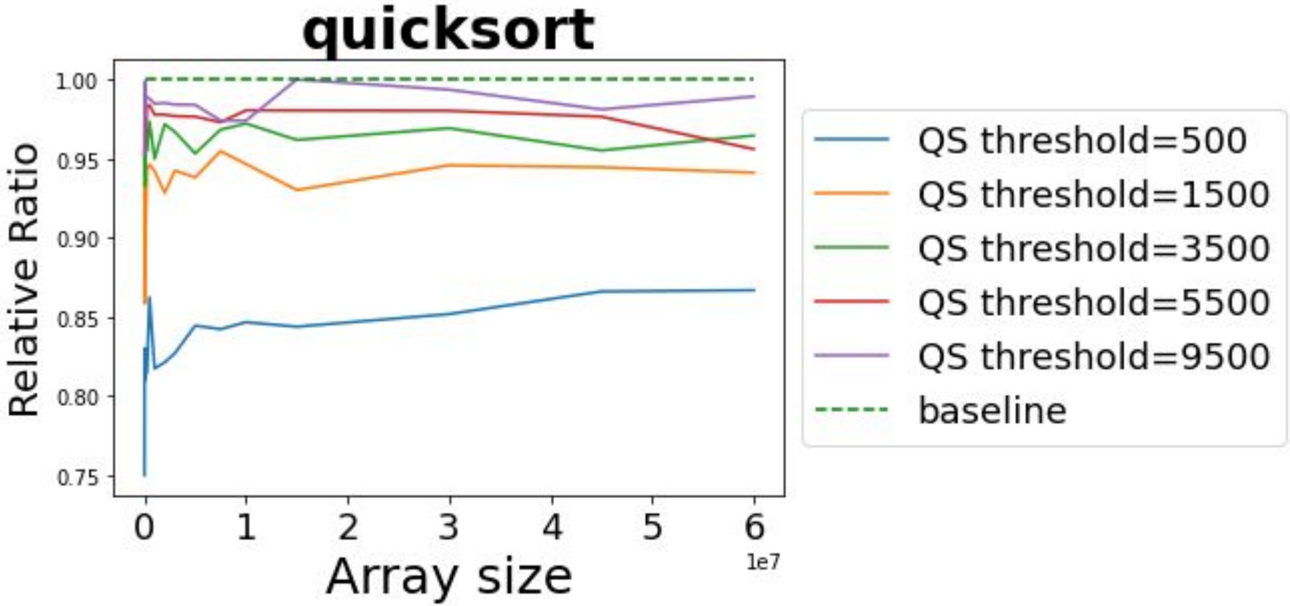
spectral-norm

Benchmarks


k-nucleotide
Relative Ratio
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
1
2
3
4
5
1e6
Sequence length

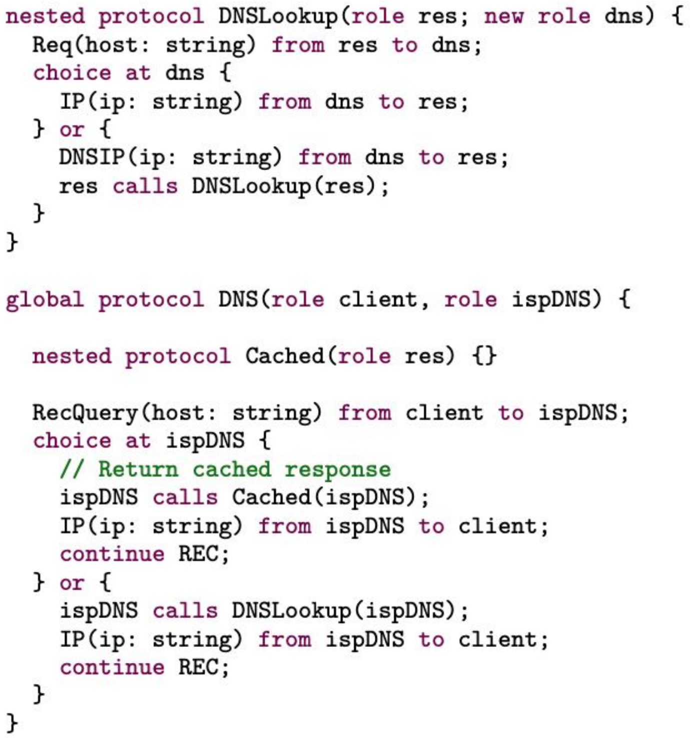

regex-redux
Relative Ratio
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.5
1.0
1.5
2.0
2.5
1e7
Sequence length

Benchmarks

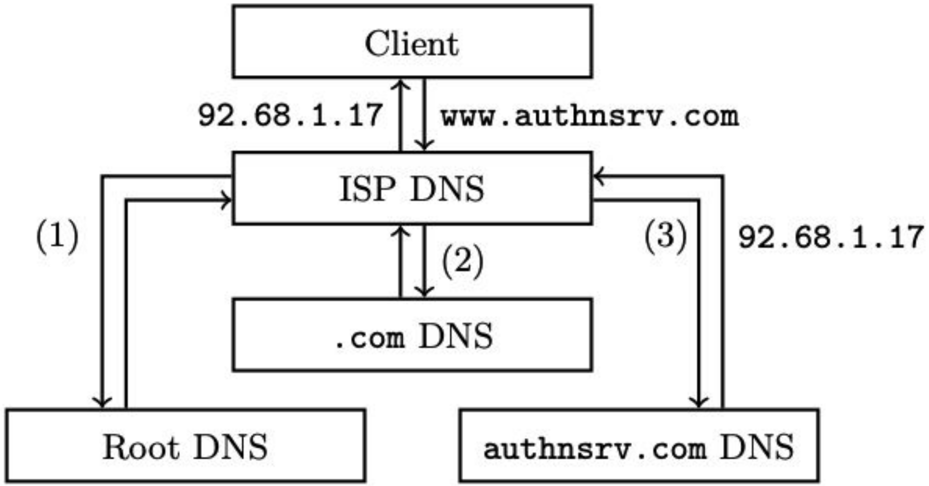

quicksort
Relative Ratio
1.00
0.95
0.90
0.85
0.80
0.75
Array size
0
1
2
3
4
5
6
1e7
QS threshold=500
QS threshold=1500
QS threshold=3500
QS threshold=5500
QS threshold=9500
baseline

Case Studies

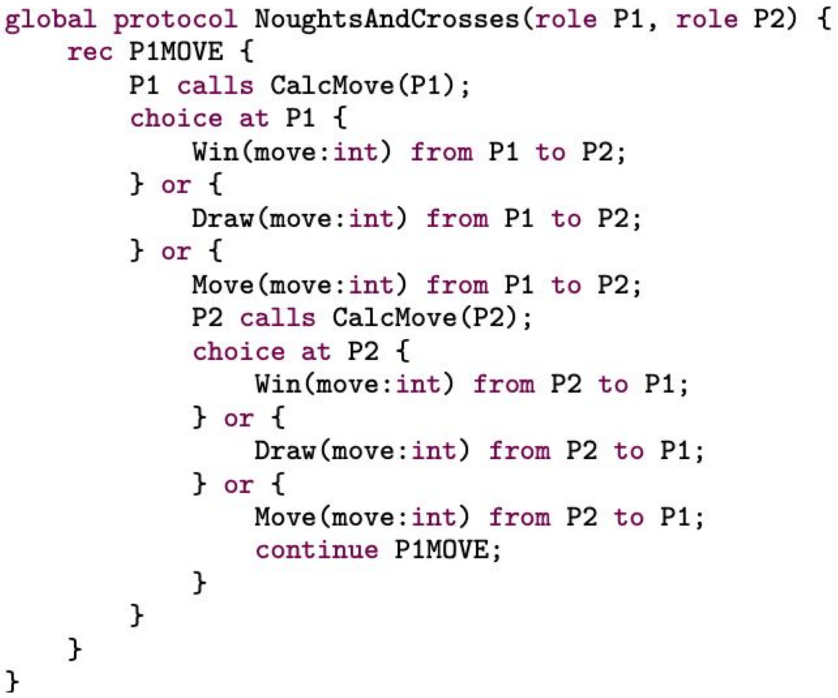
Domain Name System (DNS) Protocol



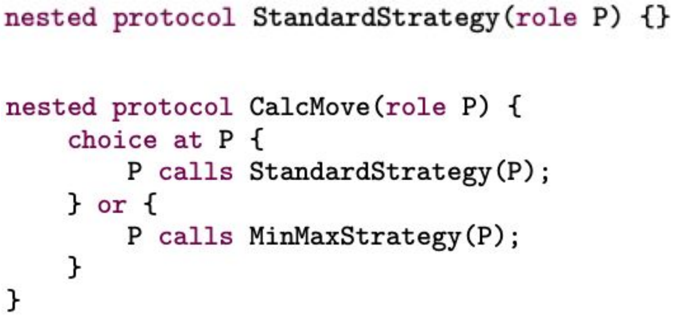
```
nested protocol DNSLookup(role res; new role dns) {
  Req(host: string) from res to dns;
  choice at dns {
    IP(ip: string) from dns to res;
  } or {
    DNSIP(ip: string) from dns to res;
    res calls DNSLookup(res);
  }
}

global protocol DNS(role client, role ispDNS) {

  nested protocol Cached(role res) {}

  RecQuery(host: string) from client to ispDNS;
  choice at ispDNS {
    // Return cached response
    ispDNS calls Cached(ispDNS);
    IP(ip: string) from ispDNS to client;
    continue REC;
  } or {
    ispDNS calls DNSLookup(ispDNS);
    IP(ip: string) from ispDNS to client;
    continue REC;
  }
}
```



Min-Max Noughts and Crosses AI



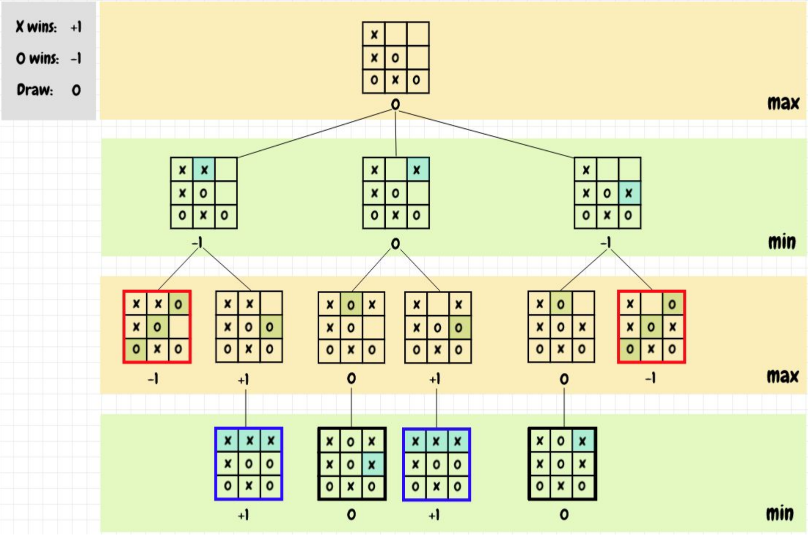
```
global protocol NoughtsAndCrosses(role P1, role P2) {
    rec P1MOVE {
        P1 calls CalcMove(P1);
        choice at P1 {
            Win(move:int) from P1 to P2;
        } or {
            Draw(move:int) from P1 to P2;
        } or {
            Move(move:int) from P1 to P2;
            P2 calls CalcMove(P2);
            choice at P2 {
                Win(move:int) from P2 to P1;
            } or {
                Draw(move:int) from P2 to P1;
            } or {
                Move(move:int) from P2 to P1;
                continue P1MOVE;
            }
        }
    }
}
```



```
nested protocol StandardStrategy(role P) {}


nested protocol CalcMove(role P) {
    choice at P {
        P calls StandardStrategy(P);
    } or {
        P calls MinMaxStrategy(P);
    }
}
```

Min-Max Noughts and Crosses AI

```
nested protocol MinMaxStrategy(role M; new role W) {
  nested protocol EvalBoard(role W) {}

  choice at M {
    CurrState(board:[]int, currP:int, toMove:int) from M to W;
    M calls MinMaxStrategy(M);
    choice at W {
      W calls MinMaxStrategy(W);
      Score(score:int) from W to M;
    } or {
      W calls EvalBoard(W);
      Score(score:int) from W to M;
    }
  } or {
    FinalState(board:[]int, currP:int, toMove:int) from M to W;
    choice at W {
      W calls MinMaxStrategy(W);
      Score(score:int) from W to M;
    } or {
      W calls EvalBoard(W);
      Score(score:int) from W to M;
    }
  }
}
```



Summary

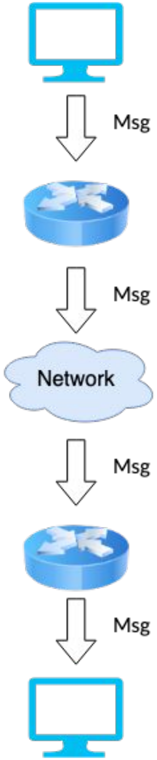
- We designed and implemented extension to the **NuScr** framework, **GoScr**[1]
  - It is the first practical implementation of MPST with **unbounded participants**
  - It can express **common programming patterns** in **Go**
  - We show that GoScr can represent **real-world protocols**
- GoScr is **more expressive** than previous work (e.g. [POPL '19])
- GoScr has **negligible performance overhead** for computationally heavy benchmarks

[1] https://github.com/nuscr/nuscr

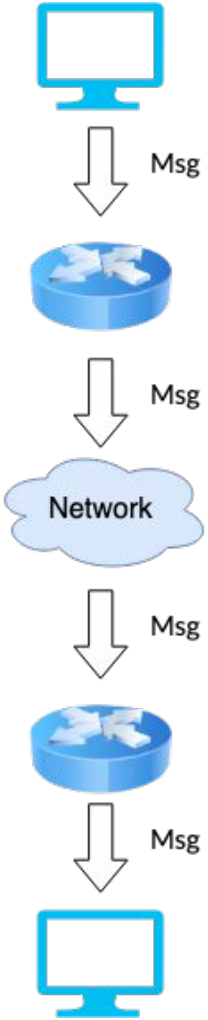
Extra Slides

Routing Protocol

```
nested protocol Forward(role Sender, role Receiver; new role
    Router) {
    Msg(int) from Sender to Router;
    choice at Router {
        Router calls Forward(Router, Receiver);
    } or {
        Msg(int) from Router to Receiver;
    }
}

global protocol Routing(role Start, role End) {
    Start calls Forward(Start, End);
}
```



Routing Protocol Demo



Expressiveness of Nested Protocols

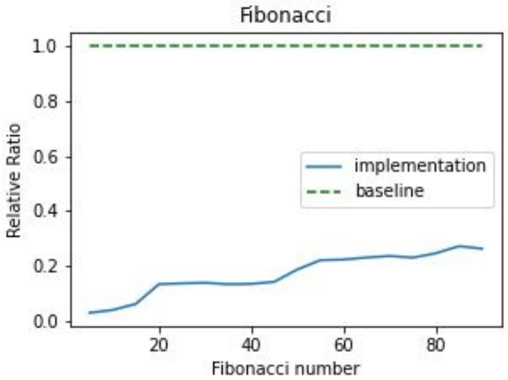
- In nested protocols, the number of participants within a protocol are **finite** and **cannot change**
  - New participants introduced through **nested protocol calls**
- Can only express processes where each step of the computation only involves a fixed number of participants
  - Can express a protocol to calculate the infinite fibonacci sequence
  - Cannot express protocols such as the unbounded primesive

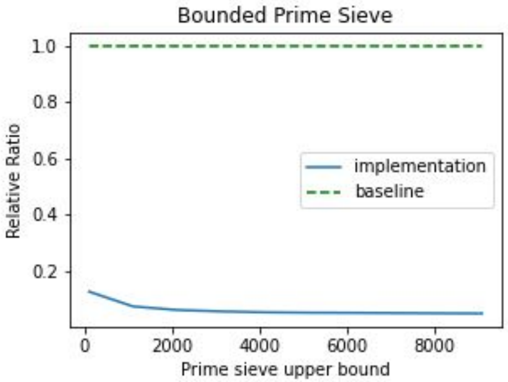
Expressiveness of Nested Protocols

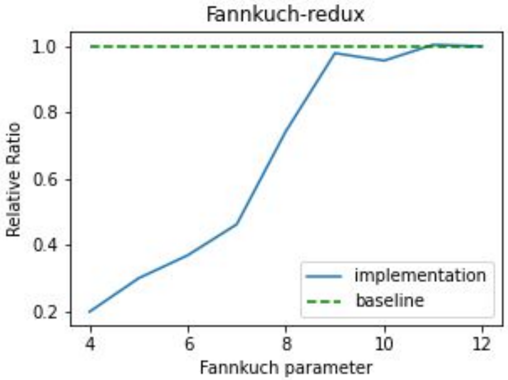
Protocol	Nested Protocols	POPL 2019
`Dynamic Ring`	✓	✗
`Dynamic Pipeline`	✓	✗
`Dynamic Fork-Join`	✓	✗
`Recursive Fork-Join`	✓	✗
`Fibonacci`	✓	✓
`Unbounded Fibonacci sequence`	✓	✗
`Fannkuch-redux`[1]	✓	✓
`Bounded Prime Sieve`	✓	✗
`Unbounded Prime Sieve`	✗	✗

[1] The Computer Language Benchmarks Game

Performance Evaluation


Fibonacci


Bounded Prime Sieve


Fannkuch-redux

- **Benchmark**
  - Speedup (t_1/ t_2) of **Scribble** (t_2) vs native Go (t_1)
  - Intel i7- 6700 processor and 16GB RAM

Contributions

- Designed and implemented extension to the **Scribble** framework[1]
  - First practical implementation of **nested session types**
  - Express **common programming patterns** in **Go**
  - Express large number of **real-world protocols**
- Compared **expressiveness** of our extension against previous work [POPL ‘19]
- Performance evaluation using a **benchmark**

[1] https://github.com/nuscr/nuscr
https://github.com/becharrens/nuscr (fork of repository)

Future work

- Prove the **correctness** of our implementation
- **Reduce overheads** of nested protocol calls
- Implement nested protocols in a **distributed setting**
- Guaranteeing **termination** in nested protocols
- Implementing nested protocols using **CFSMs**

Scope of Protocols

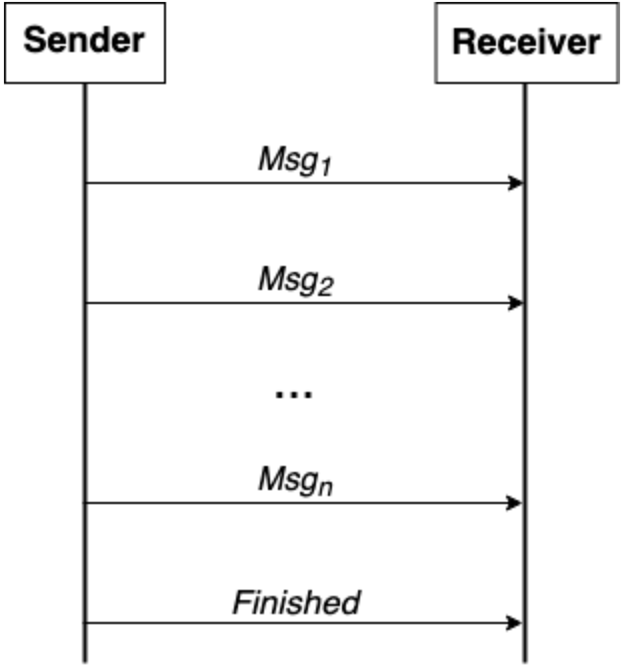
- Top-level scope
- Every protocol introduces its own scope
- Protocols defined within a scope cannot be accessed outside that scope
- Allow **shadowing** of protocol names
  - Declaration of a protocol with the same name in a subscope overrides previous definition

Renaming protocols

- **Flatten** structure of Scribble module
    - Resolve name clashes between nested protocols in different scopes
    - Resolve name clashes between global and nested protocols
- Generate **unique names** for each protocol
- Update references in protocol calls
- Simplifies definition of projection
- Needed for code generation

Recursion

- Difficult to design a correct implementation for protocols combining:
  - Asynchronous communication
  - Choice
  - Recursion



Recursion – Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}


func Sender(sendChan chan int,
    endChan chan string) {

    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {

    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Recursion – Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}
```

```
func Sender(sendChan chan int,
    endChan chan string) {
    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {
    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Generated Output:

```
0
Finished
```

Recursion – Possible Implementation

```
func main() {
    numChan := make(chan int, 100)
    endChan := make(chan string, 1)
    go pipeline.Sender(numChan,
endChan)
    go pipeline.Receiver(numChan,
endChan)
    time.Sleep(1 * time.Second)
}
```

```
func Sender(sendChan chan int,
    endChan chan string) {

    for i := 0; i < 100; i++ {
        sendChan <- i
    }
    endChan <- "Finished"
}
```

```
func Receiver(recvChan chan int,
    endChan chan string) {

    for {
        select {
        case num := <-recvChan:
            fmt.Println(num)
        case endMsg := <-endChan:
            fmt.Println(endMsg)
            return
        }
    }
}
```

Race Condition

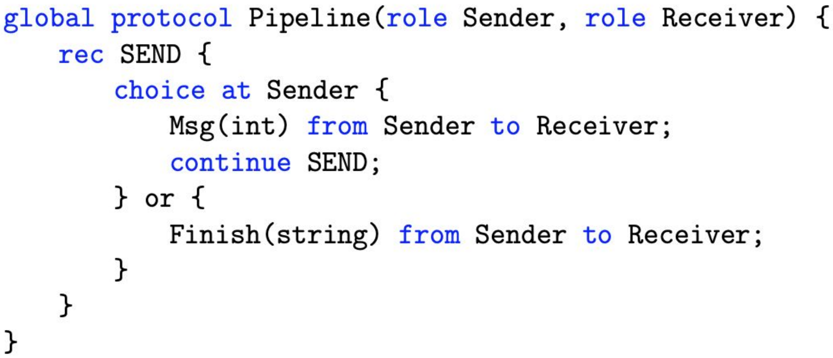
Channels are reused throughout all the choices

Extracting Recursion into Protocols

- **Reusing channels** in different unfoldings of recursion leads to **race conditions**
- Cannot allocate all necessary channels **statically**
  - Potentially infinite recursion unfoldings
- **Allocate channels dynamically** at the beginning of each unfolding of the recursion
  - **Generate** new **protocols** with the body of each recursion

Recursion Extraction

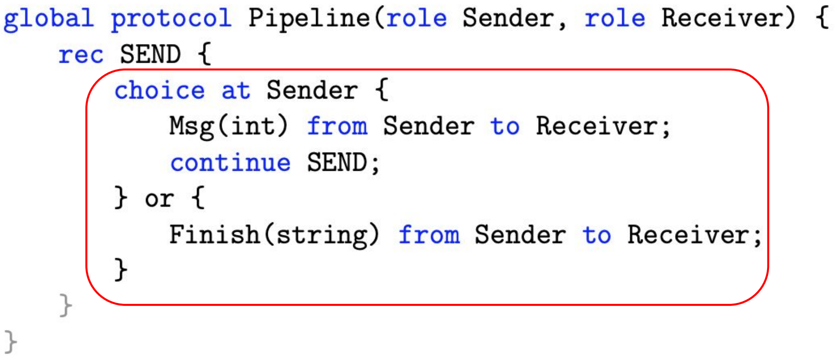
Before extraction



```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

Recursion Extraction

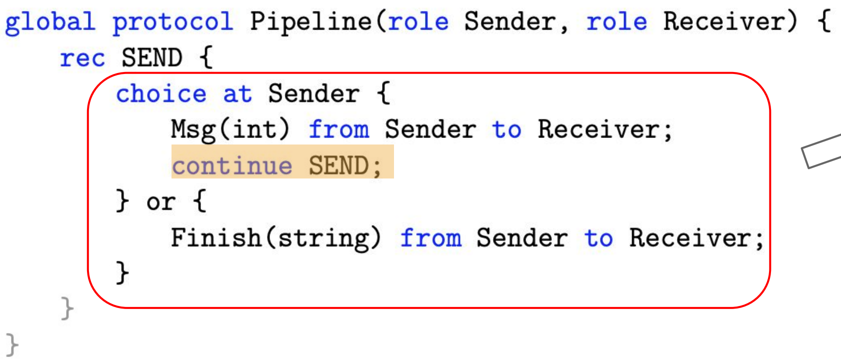
Before extraction



```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

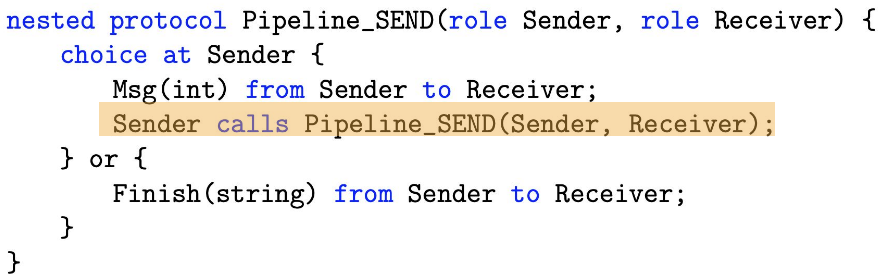
Recursion Extraction

Before extraction



```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

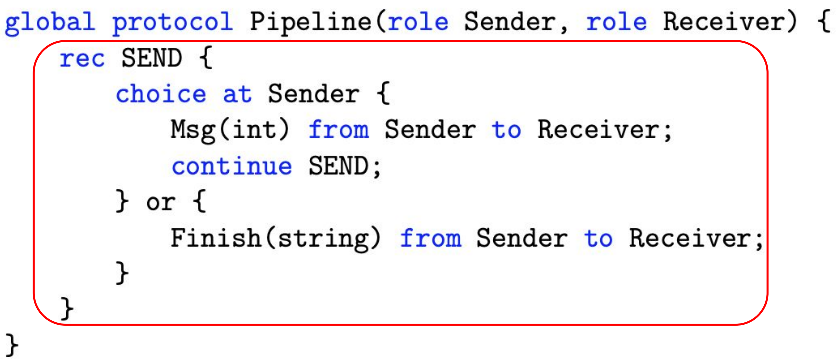
After extraction



```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

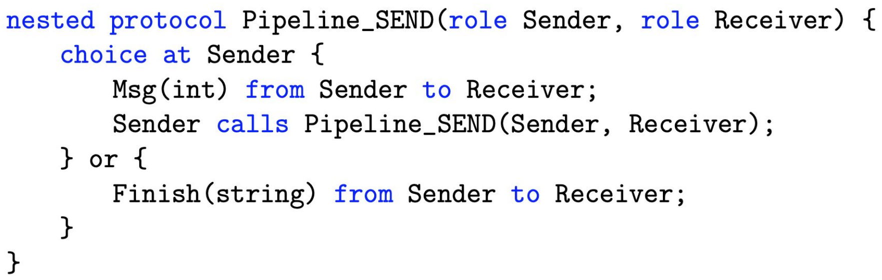
Recursion Extraction

Before extraction



```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction



```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

Recursion Extraction

Before extraction

```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

```
global protocol Pipeline(role Sender, role Receive) {
    Sender calls Pipeline_SEND(Sender, Receive);
}
```

Recursion Extraction

Before extraction

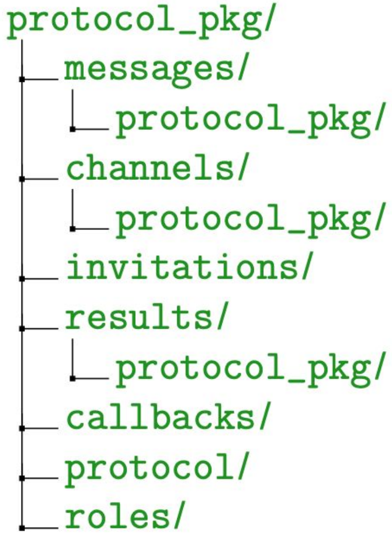
```
global protocol Pipeline(role Sender, role Receiver) {
    rec SEND {
        choice at Sender {
            Msg(int) from Sender to Receiver;
            continue SEND;
        } or {
            Finish(string) from Sender to Receiver;
        }
    }
}
```

After extraction

```
nested protocol Pipeline_SEND(role Sender, role Receiver) {
    choice at Sender {
        Msg(int) from Sender to Receiver;
        Sender calls Pipeline_SEND(Sender, Receiver);
    } or {
        Finish(string) from Sender to Receiver;
    }
}
```

```
global protocol Pipeline(role Sender, role Receive) {
    Sender calls Pipeline_SEND(Sender, Receive);
}
```

Implementation Structure



```
protocol_pkg/
├── messages/
│   └── protocol_pkg/
├── channels/
│   └── protocol_pkg/
├── invitations/
├── results/
│   └── protocol_pkg/
├── callbacks/
├── protocol/
└── roles/
```

Package messages

- ➔ Generate structs for the different labeled messages exchanged in the protocol
- ➔ Fields in struct correspond to payload of the message

```go
type Msg struct {
    Int int
}
```

Package channels

- Channels used by the roles for labeled message exchanges are stored in a struct
- Each channel will only be used in one exchange

```go
type Router_Chan struct {
    Receiver_Msg chan forward.Msg
    Sender_Msg chan forward.Msg
}
```

Package invitations

- Each role has a struct storing all the channels needed to send and receive invitations
- Invitations consist of:
    - Channel struct
    - Invitation struct

```
type Forward_Router_InviteChan struct {
	Invite_Receiver_To_Forward_Receiver chan forward.Receiver_Chan
	Invite_Receiver_To_Forward_Receiver_InviteChan chan Forward_Receiver_InviteChan
	Invite_Router_To_Forward_Sender chan forward.Sender_Chan
	Invite_Router_To_Forward_Sender_InviteChan chan Forward_Sender_InviteChan
}
```

Package callbacks

- ➔ Protocol logic implemented through callbacks
  - ◆ Callback calls interleaved in role implementation
- ➔ Define **interface** with methods that define a role’s behaviour, which the user must implement

```
type Forward_Router_Env interface {
    Msg_To_Receiver() forward.Msg
    Done()
    ResultFrom_Forward_Sender(result
forward_2.Sender_Result)
    To_Forward_Sender_Env()
Forward_Sender_Env
    Forward_Setup()
    Router_Choice() Forward_Router_Choice
    Msg_From_Sender(msg forward.Msg)
}
```

Package results

- **Non-dynamic participants** in a protocol will generate a result
  - Mechanism for returning results of computation in the protocol outside of the session
- Generate empty struct - user defines what useful information should be returned

```
type Sender_Result struct {

}
```

Contributions

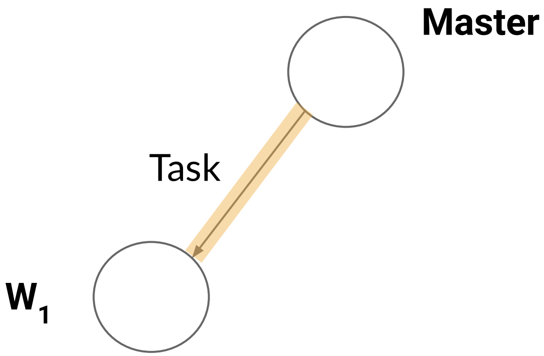
- Extended MPST-based framework so it can statically verify the specification of nested protocols
- Developed **first practical application** of **nested protocols** theory
    - Increased Scribble's expressiveness with the ability to model many real-world applications
- Generate correct implementations in Go using its inbuilt concurrency primitives
- Proposed approach to return results from nested subsessions

Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```

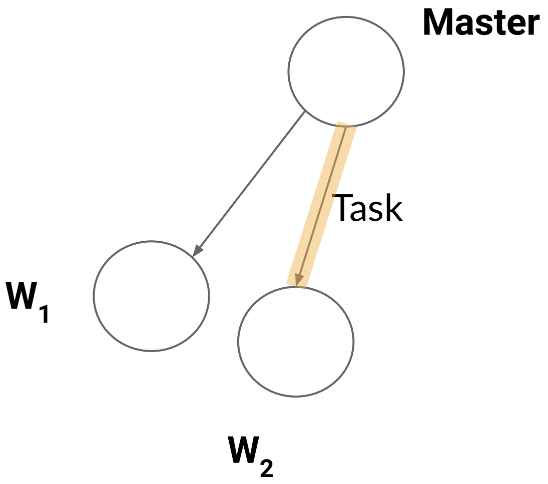
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



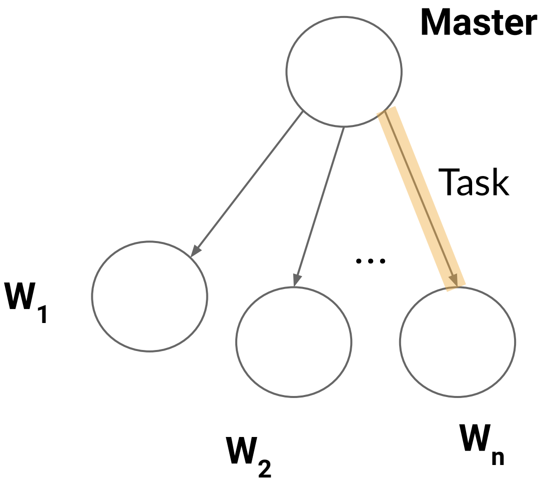
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



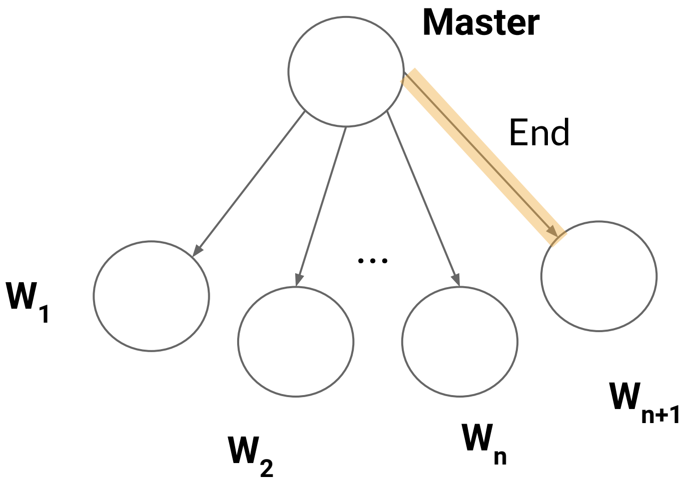
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



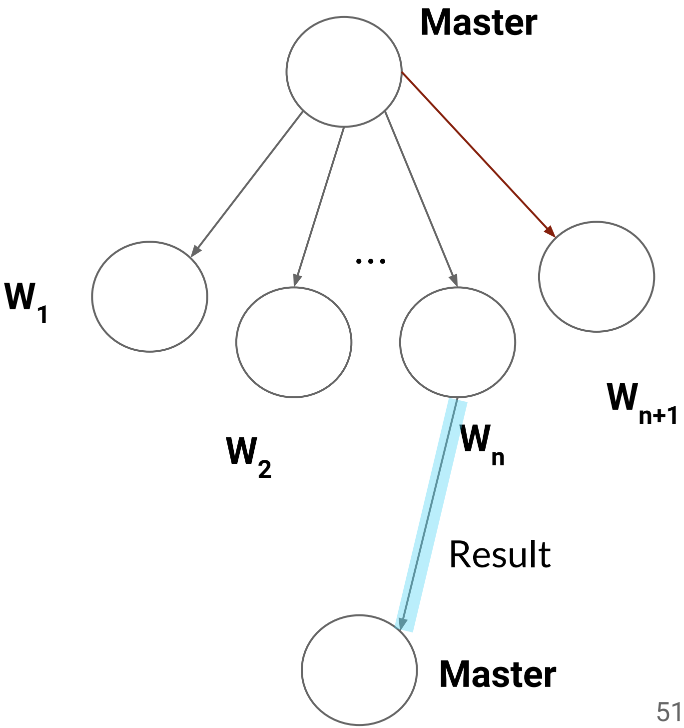
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



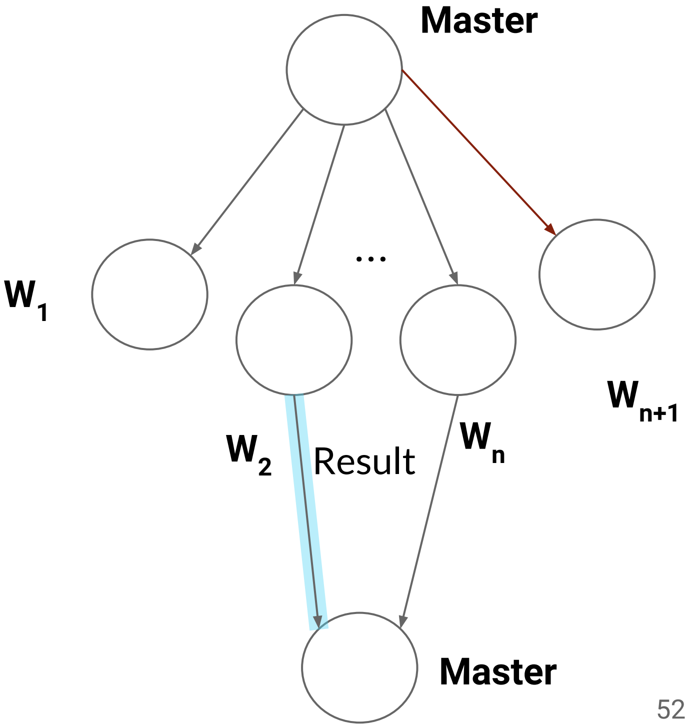
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



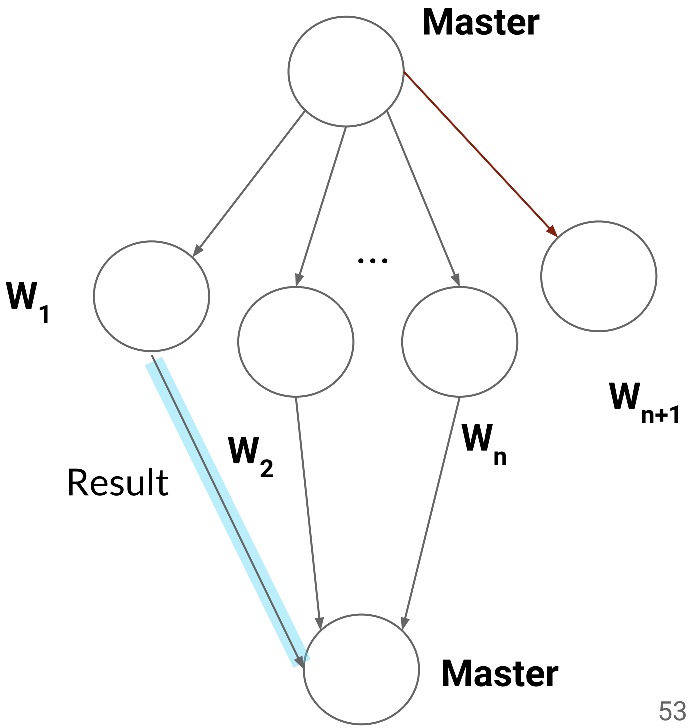
Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



Fork-Join

```
nested protocol Fork(role M; new role W) {
    choice at Master {
        Task() from M to W;
        M calls Fork(M);
        Result() from W to M;
    } or {
        End() from M to W;
    }
}

global protocol ForkJoin(role Master, role Worker) {
    choice at Master {
        Task() from Master to Worker;
        Master calls Fork(Master);
        Result() from Worker to Master;
    } or {
        End() from Master to Worke;
    }
}
```



Code Generation Approach

- Generate role APIs from their **local protocols**
  - Implementation is **correct by construction**
- Roles execute as **goroutines** which communicate over **shared memory channels**
- Protocol implementation defined through **callbacks**
- Role implementation **returns result**