Implementing Multiparty Session Types in Rust

Nicolas Lagaillardie
Imperial College London

Rumyana Neykova
Brunel University London

Nobuko Yoshida
Imperial College London

Multiparty Session Types (MPST) is a typing discipline for communication protocols, which ensures communication safety and deadlock-freedom for more than two participants. This paper reports our on-going research project, implementing multiparty session types in Rust. Current Rust implementations of session types are limited to binary (two party communications) and introduce errors due to a gap between affinity in Rust and linearity in the session types discipline. To achieve our goal by overcoming these limitations, we extend an existing library for the binary session types to MPST. We created an environment involving a server and clients, communicating through a MQTT (MQ Telemetry Transport) broker with protocols that have been checked by our library.

1 Introduction

In the last decade, the software industry has seen a shift towards programming languages that promote the coordination of concurrent and/or distributed software components through the exchange of messages over *communication channels*. Languages with native message-passing primitives (e.g., Go, Elixir and Rust) are becoming increasingly popular. In particular, Rust has been named the most loved programming language in the annual Stack Overflow survey for four consecutive years (2016-19)[1].

The advantages of message-passing concurrency are well-understood, i.e it allows cheap horizontal scalability at a time when technology providers have to adapt and scale their tools and applications to various devices and platforms. Message-passing based software, however, is as liable to errors as other concurrent programming techniques [12]. Much academic research has been done to develop rigorous theoretical frameworks for verification of message-passing programs. One such type of theoretical framework is *multiparty session types* (MPST) [4] – a type-based discipline that ensures that concurrent and distributed systems are *safe by design*. It guarantees that message-passing processes following a predefined communication protocol, are free from communication errors and deadlocks.

Rust is a particularly appealing language for the practical embedding of session types. Its *affine type system* allows for static typing of linear resources – an essential requirement for the safety of session type systems. Rust combines efficiency with message-passing abstractions, thread and memory safety [8], and has been used for the implementation of large-scale concurrent applications such as the Mozilla browser, Firefox, and the Facebook blockchain platform, Libra. Despite the interest in the Rust community for verification techniques handling multiple communicating processes[2], the existing Rust implementations [7, 9] are limited to *binary*, i.e., two party session types.

In this paper, we give an overview of the state-of-the-art Rust implementations, discuss the challenges of scaling the implemented libraries to multiparty protocols, and present our preliminary design for multiparty session types in Rust. Our design follows a state-of-the-art encoding of multiparty into binary session types [11]. We propose to generate local APIs in Rust, utilising the Scribble toolchains. The generated APIs can be built on top of an existing binary session types library by applying only a

[1] `https://insights.stackoverflow.com/survey/2019`
[2] `https://users.rust-lang.org/t/anybody-working-on-multiparty-session-types-for-rust/10610`

Submitted to:
PLACES 2020

minor modification to the type signatures of the underlying primitives. Differently from other MPST implementations that check at runtime the linear usage of channels (e.g. [11, 5]), we rely on the Rust affine type system to type-check MPST programs.

In addition, since we generate the local types from a readable global specification, errors caused by an affine (and not linear) usage of channels, a well-known limitation of the previous libraries, are easily avoided.

This paper is organised as follows. We first detail the different existing solutions, and their limitations in § 2. In § 3 we present the real case scenario which pushed us to work on multiparty session types in Rust. Finally, § 4 illustrates our preliminary design. § 5 concludes with related and future work.

2 Limitations of Existing Implementations

We first summarise two implementations of session types in Rust – *Session Types for Rust* by Munksgaard *et al* [7], and *Rusty Variation, Deadlock-free Sessions with Failure in Rust* by Kokke [9].

The library presented in [7] implements binary session types, following [3]. It checks at compile-time that the behaviours of two endpoint processes are *dual*, i.e the processes are compatible. The library in [9], based on the formal EGV calculus by Fowler *et al* [2], allows to write and check session typed communications, and additionally support exception handling constructs. Both libraries include the four basic communication primitives: *send*, *receive*, and *internal* and *external choice*. Rust originally did not support *recursive types* so that the work [7] had to uses de Bruijn indices to encode recursive session types, while the work [9] uses Rust's native recursive types. Although both libraries share the same goal, the underpinning implementations differ significantly.

The main difference in the implementation of the two libraries is the treatment of *failure handling* when protocols are closed prematurely. This is also one of the main technical challenges when implementing session types in an affine language. An affine type system stipulates that a resource (a value) is used *at most once*, while a linear type system, such as the type system of session types, requires that a value is used *exactly once*. A naive implementation of session types in Rust may allow for channel values to be dropped *at any time*, i.e before the protocol has been completed. To prevent dropping a channel prematurely, [7] utilises a succession of *destructor bombs* and `std::mem::forget`[3] to ensure that a channel value is never dropped. As a result, if a channel is dropped, the library *panics*, causing a `segmentation fault`. A disadvantage of this approach is that it leads to a memory leak. The second library [9] prevents this behaviour by customising the native destructor `Drop` in Rust. When a session's destructor is called, the session is first disconnected, dropping every channel value used in the session, then the memory is deallocated.

Another minor implementation difference between [7] and [9] is the management of external and internal `choice`. In [7], different choice branches are encoded as numbers, which is error-prone. Kokke's library [9] suggests a more usable solution, where choice branches are distinguished based on label types. More precisely, the work [9] implements the sum type `Either`, and the macros `choose` and `offer` which generalises binary choice using `Either` to a choice on any enum.

Finally, both libraries suffer from a well-known limitation of binary session types[4]. Notably, since deadlock-freedom is ensured only inside a session, a Rust endpoint process, that communicates with more than one other process, is prone to deadlocks and communication errors. MPST solves that limitation by expanding the scope of a session to multiple participants. In the rest of the paper, we present our

[3] `https://doc.rust-lang.org/1.35.0/std/mem/fn.forget.html`

[4] `https://github.com/Munksgaard/session-types/commit/0f25ccb7c3bc9f65fa8eaf538233e8fe344a189a`

preliminary design of MPST in Rust. We build on top of the library in [9] since, as already explained, it offers several improvements in comparison to [7].

3 Use case: Amazon Prime Video Protocol in MPST

The Amazon Prime Video streaming service is a use case which can take full advantage of multiparty session types. Each streaming application connects to servers, and possibly other devices, in order to access services, and follows some specific protocol.

To present our design, we follow a simplified version of the protocol, illustrated in the diagram in Figure 1. The protocol involves three services – an `Authenticator` service, a `Server` and a `Client`. At first, `Client` connects to `Authenticator` by providing an identifying ID. If the ID is accepted, the session continues with a choice on `Client` to either request a video or end the session. The first branch is, *a priori*, the main service provided by Amazon Prime Video. `Client` cannot directly request videos from `Server`, and has to go through `Authenticator` instead. On the diagram, the choice is denoted as the frame `alt` and the choices are separated with the horizontal dotted line. The protocol is recursive, and `Client` can request new videos as many times as needed. The arrow going back on `Client` side in Figure 1 represents this recursive behaviour. To end the session, `Client` should first send `Close` message to `Authenticator`, which then subsequently sends a `Close` message to `Server`.

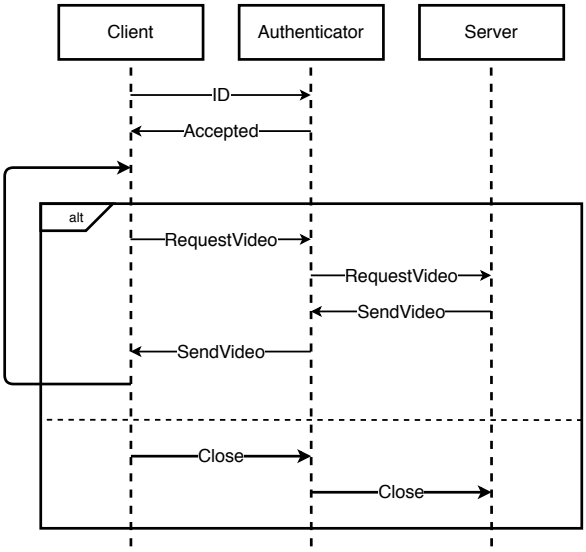

Figure 1: Workflow for Amazon Prime Video Use-case

4 Methodology and Design: Multiparty Session Types in Rust

The top-down methodology of multiparty session types, is illustrated in Figure 2. It follows three main steps [13, 4]. First, a *global type*, also called a *global protocol*, is defined as a shared contract between communicating endpoint processes. Scribble (a protocol description language for multiparty session types) [14] provides facilities for writing and verifying global protocols. A global protocol is then *projected* to each endpoint process, resulting in a *local type*. A local type involves only

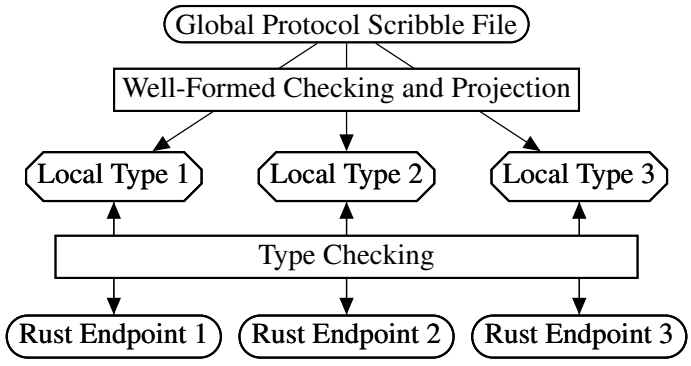

Figure 2: Multiparty Session Types Methodology

```
1  module BasicProtocol; // Name of the module
2
3  type <Rust> "Id" from "port::Id" as Id; // Definition of the Id type in Rust
4  type <Rust> "Answer" from "port::answer" as Answer; // Definition of the Answer type in Rust
5  type <Rust> "Video" from "port::videos" as Video; // Definition of the Video type in Rust
6  type <Rust> "String" from "std::string::String" as string; // Definition of the String type in Rust
7
8  global protocol VideoStreamingProtocol(role Auth, role Client, role Server) {
9    Declare(Id) from Client to Auth; // Sends a message labelled Declare with a payload Id
10   Accept(Answer) from Auth to Client; // The Auth role replies with an Accept message
11   do VideoRequestProtocol(Auth, Client, Server); // Call the VideoRequestProtocol
12   rec Loop {
13    choice at Client { // Client makes a choice
14     RequestVideo(string) from Client to Auth; // Client sends a request for a video, giving its name
15     RequestVideo(string) from Auth to Server; // Auth forwards the request
16     SendVideo(Video) from Server to Auth; // Server sends the video file to the Auth
17     SendVideo(Video) from Auth to Client; // Auth sends the video file to the client
18     continue Loop; // A Recursive call
19   } or {
20     Close() from Client to Auth; // Close the session between Client and Auth
21     Close() from Auth to Server; // Close the session between Server and Auth
22   }
23 }
```

Listing 1: Main Body of the Scribble Protocol.

the interactions specific to a given endpoint. Finally, each endpoint process is type-checked against its projected local type. Type-checking is done either statically, dynamically or a mixture of both (called hybrid in [5]). In this section, we explain how the framework of MPST can be applied to Rust.

4.1 Global Protocol with Scribble

Scribble is a protocol language for describing global protocols, which can also validate their well-formedness [14]. The global protocol of our running example written in Scribble is presented in Listing 1.

Scribble protocols are organised into *modules*, divided in the declaration of *message payload types*, and one or more global protocol definitions. Here, our module is called *BasicProtocol*, declared at line 1 using the keyword `module`. The different message payload types are declared on lines 3 –5 and corresponds to the types *Id* (line 3), *Answer* (line 4) and *Video* (line 5). Note that these declarations point to the actual type declaration in Rust. The global protocol, `VideoStreamingProtoco`, is parameterised on three roles, related to the three processes involved in our use case. In Scribble, the syntax for peer-to-peer message exchange is written as `Label(payload) from A to B` where `Label` is an identifying label for the message, and the `payload` is a list of the types of the payloads of the message, `A` is a sending role and `B` is a receiving role. For instance, at line 9, `Client` role sends a message labelled `Declare` containing the payload `Id` to `Authenticator` role. As we already explained, each payload type is mapped to an actual Rust type. At line 12, the statement `rec Loop` denotes the start of a recursive block.

4.2 From Binary to Multiparty Session Types

We extend the library for binary session types to handle multiparty communication. Following the methodology in Figure 2, our global protocol is *projected* into the three local types given in Listing 2.

```
local protocol VideoProt at A(...)
{ Declare(Id) from C;
 Accept(Answer) to C;
 rec Loop {
  choice at Client {
   RequestVideo(string) from C;
   RequestVideo(string) to S;
   SendVideo(Video) from S;
   SendVideo(Video) to C;
   continue Loop;
 } or {
   Close() from C;
   Close() to S;
}}
```

```
local protocol VideoProt at C(...)
{ Declare(Id) to A;
  Accept(Answer) from A;
  rec Loop {
   choice at Client {

    RequestVideo(string) to A;
    SendVideo(Video) from S;

    continue Loop;
   } or {

    Close() to A;
}}
```

```
local protocol VideoProt at S(...)
{

 rec Loop {
  choice at C {

   RequestVideo(string) from A;
   SendVideo(Video) to C;

   continue Loop;
  } or {

   Close() from A;
}}
```

Listing 2: Local Protocols

To extend binary session types to multiparty protocols, we introduce the notion of a *role*. In binary session types, *roles* are optional as there are only one sender and one receiver involved in the same session. In multiparty session types, it is necessary to distinguish the different processes involved in the same session. Roles abstract the low-level representation of the communication channels. Hence, roles are mapped to channels corresponding to the underlying transport, for example, they can be mapped to a TCP socket, a shared memory cell, an MQTT socket, etc. Taking the new Role types into account, we have modified the basic communication primitives of the session type API presented in [9]. The updated MPST API is given below.

```
pub struct Role { name: string, }
pub struct Send<T, R: Role, S: Session> { ... }
pub struct Recv<T, R: Role, S: Session> { ... }
pub struct End<> { ... }
```

```
pub fn send(x: T, r: Role, s: Send<T, R, S>) -> S
pub fn recv(r:Role, s: Recv<T, R, S>) -> Option<(T, S)>
pub fn close(s: End) -> Option<()>
```

Listing 3: MPST Rust API

We can generate the specific API for each endpoint process from a global protocol given in Scribble, utilising the MPST communication primitives defined in Listing 3. Listing 7 shows the syntax of the (generated) local types for the global protocol in Listing 1. The types are defined as the bottom up, *i.e.*, the type corresponding to the last protocol interaction is defined first. For example, the type for the server role is defined by first specifying the last operation, `Send` the video to role `A`, then the previous operation, *i.e.*, `Recv` a request video from the `A`, is defined. The Authenticator and the Client's local types are defined similarly, with the addition of the offer type, declared as `enum`.

```
// Declaration of the roles
const A = Role::new("Authenticator");
const C = Role::new("Client");
const S = Role::new("Server");

// Server's local type
type SendVideoS= Send<string, A, End<>>;
type RequestVideoS= Recv<string, A, SendVideoS>;
```

```
// Client's local type
type SendVideoA= Recv<string, A, ClientOffer>;
type RequestVideoA= Send<string, A, SendVideoA>;
type EndC= End<>;
type CloseC= Send<string, A, EndC>;
enum ClientOffer { RVC(A, RequestVideoA), CAC(A, CloseC)};
type ClientAccept= Recv<string, A, ClientOffer>;
type ClientConnection= Send<string, A, ClientAccept>;
```

Listing 4: Roles and Local Types in Rust (generated from Scribble)

```
fn server(s: RequestVideoS) -> Option<()> {
 let (id, s) = recv(A, s)?;
 let s = send("video", A, s);
 close(s);
}}}

fn client(c: ClientConnection) -> Option<()> {
 let c= send("id", A, c);
 let (answer, c) = recv(A, c)?;
 choiceClient(c);
}
```

```
fn choiceClient(c: ClientAccept) -> Option<()> {
 choose!(c, {ClientOffer::RVC(A, c) => {
  let a = send(title, A, c);
  let (video, a) = recv(A, c)?;
  choiceClient(a);
 }, ClientOffer::CAC(A, c) => {
  close(c);
}}}
```

Listing 5: Server and Client Endpoint

```
fn authenticator(a: AuthConnection) -> Option<()> {
  let (id, a) = recv(C, a)?;
  let a = send("accepted", C, a);
  choiceAuthenticator(a);
}

fn offerAuthenticator(a: AuthAccept) -> Option<()> {
  offer!(a, {AuthenticatorOffer::RVC(C, a) => {
   let (title, a) = recv(C, a)?;
  let a = send(title, S, a);
  let (video, a) = recv(S, a)?;
  let a = send(video, C, a);
  offerAuthenticator(a);
 }, AuthenticatorOffer::CAC(C, a) => {
  close(a);
}}}
```

Listing 6: Authenticator Endpoint

```
// Authenticator's local type
// build from the bottom up
type SendVideoC = Send<string, C, AuthOffer>;
type SendVideotS = Recv<string, S, SendVideoC>;
type RequestVideoS = Send<string, S, SendVideoS>;
type RequestVideoC = Recv<string, C, RequestVideoS>;
type EndAuthConnection = End<>;
type CloseA = Recv<string, C, EndAuthConnection>;
enum AuthOffer {
  RVC(C, RequestVideoC), CAC(C, CloseA)};
type AuthAccept = Send<string, C, AuthOffer>;
type AuthConnection = Recv<string, C, AuthAccept>;
```

Listing 7: Authenticator Types

Listing 6 shows the endpoint process implementation for the Authenticator role. The endpoint process is type-checked against the local types, given in Listing 7. The process is checked at compile-time and the detected errors, if any, will be displayed directly in the Cargo console (the editor for Rust programming). The above implementation, although intuitive, does not resolve the inherent conflict between Rust, which is affine, and session types, which is linear. The implementation suffers from the same drawback as the binary session types API in [9]. However, the MPST methodology is a step forward to w.r.t usability. Differently than the Rust local types which can get convoluted, the syntax of global protocols is user-friendly and readable. Developers can use the global protocol as a guidance, and hence avoid errors such as prematurely ending of a session. Moreover, as observed in [9], most of the errors are caused by misuse of methods and functions. Since we are code-generating the local types, the chance of misspelling is significantly reduced. Another option to ensure that a protocol is fully implemented is to utilise Rust's type inference and to compare if the inferred channel type is compatible with the generated one. Note that this approach cannot work with the types presented in Listing 7; it requires the generation of chained APIs, as in [5].

4.3 Distributed Execution Environment

To test our simple example in a more realistic environment, we have explicitly created *devices* on Rust, connected through MQTT (MQ Telemetry Transport) [6]. MQTT is a messaging middleware for ex-

changing messages between devices, predominantly used in IoT networks. It is lightweight and incurs no significant bandwidth overhead, which makes it an ideal transport for the execution of highly distributed protocols between multiple devices. We have configures the MQTT protocol to run over TCP/IP, with an MQTT server, called MQTT broker, hosted on `https://www.cloudmqtt.com/`. At the start of the protocol, each device connects to a public MQTT channel, and a session is established. Once a session is established, the channel id of the connected device is mapped to the role of the endpoint process. Therefore, in a distributed case running over MQTT transport, each role is mapped to an MQTT channel.

5 Conclusion and Related and Future Work

We have shown our plan to implement multiparty session types in Rust. We gave an overview of the existing binary session type implementations in Rust, notably [7] and [9]. We plan to extend [9] to implement our new MPST library and Scribble toolchain. We proposed a simplified protocol based on a real case scenario, use it as a running example and discussed its execution in a distributed environment.

Our proposed design follows the methodology given by [5], which generates Java communicating APIs from Scribble. This, and other multiparty session types implementations, exploit the equivalence between local session types and communicating automata to generate session types APIs for mainstream programming languages (e.g., Java [5, 10], Go [1], F# [11]). Each state from state automata is implemented as a class, or in the case of [10], as a type state. To ensure safety, state automata have to be derived from the same global specification. All of the works in this category use the Scribble toolchain to generate the state classes from a global specification. All of these implementations detect linearity violations at runtime and offer no static alternative.

This paper proposes the generation of protocol-specific APIs, which promotes type checking of protocols *at compile-time*. This is done by projecting the endpoints' state space in those protocols to groups of channel types in the desired language. The resulting hybrid verification guarantees the compliance of the protocol at compile-time. This protocol is used in a distributed system built over MQTT transport. We also gave an overview of the multiparty session types, syntax and semantics we plan to implement.

Until PLACES'2020, we plan to finish the first version of the prototype for the API generation from Scribble, finalising the design of MPST local protocols and Rust APIs. In the presentation, we plan to explain how a subtle interplay between affinity in Rust and linearity of the session types discipline reflects our design decision and how our MPST methodology is useful to reduce this gap to foster discussions at the workshop. We also plan to include a demo in the presentation. At the same time, we will extend our running protocol more usable for the real use case and our live system to integrate with authentication mechanisms.

References

[1] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): *Distributed Programming Using Role Parametric Session Types in Go*. In: *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, 3, ACM, pp. 29:1–29:30.

[2] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types Without Tiers*. *Proc. ACM Program. Lang.* 3(POPL), pp. 28:1–28:29, doi:10.1145/3290341.

[3] Kohei Honda, Vasco T Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In: *European Symposium on Programming*, Springer, pp. 122–138.

[4] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. *POPL* 43(1), pp. 273–284.

[5] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification Through Endpoint API Generation*. In Perdita Stevens & Andrzej Wasowski, editors: *Fundamental Approaches to Software Engineering*, 9633, Springer Berlin Heidelberg, pp. 401–418, doi:10.1007/978-3-662-49665-7_24. Available at `http://link.springer.com/10.1007/978-3-662-49665-7_24`.

[6] Urs Hunkeler, Hong Linh Truong & Andy Stanford-Clark (2008): *MQTT-SA publish/subscribe protocol for Wireless Sensor Networks*. In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, IEEE, pp. 791–798.

[7] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session types for Rust*. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, ACM, pp. 13–22, doi:10.1145/2808098.2808100.

[8] Steve Klabnik & with contributions from the Rust Community Carol Nichols (2019): *The Rust Programming Language*, 1.35.0 edition. Available at `https://doc.rust-lang.org/1.35.0/book/`.

[9] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. In: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, pp. 48–60, doi:10.4204/EPTCS.304.4. Available at `https://doi.org/10.4204/EPTCS.304.4`.

[10] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking protocols with Mungo and StMungo*. In: *PPDP*, pp. 146–159, doi:10.1145/2967973.2968595. Available at `http://doi.acm.org/10.1145/2967973.2968595`.

[11] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In: *31st European Conference on Object-Oriented Programming*, *LIPIcs* 74, Schloss Dagstuhl, pp. 24:1–24:31.

[12] Tengfei Tu, Xiaoyu Liu, Linhai Song & Yiying Zhang (2019): *Understanding Real-World Concurrency Bugs in Go*. In: *ASPLOS*, ACM, pp. 865–878.

[13] Nobuko Yoshida & Lorenzo Gheri (2020): *A Very Gentle Introduction to Multiparty Session Types*. In Dang Van Hung & Meenakshi DSouza, editors: *Distributed Computing and Internet Technology*, Lecture Notes in Computer Science, Springer International Publishing, pp. 73–93, doi:10.1007/978-3-030-36987-3_5.

[14] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2013): *The Scribble protocol language*. In: *International Symposium on Trustworthy Global Computing*, Springer, pp. 22–41.