DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DAVID MARIA ALMEIDA AMORIM DA COSTA
Bachelor in Computer Science

SESSION KOTLIN

A HYBRID SESSION TYPE EMBEDDING IN KOTLIN

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

SESSION KOTLIN

A HYBRID SESSION TYPE EMBEDDING IN KOTLIN

DAVID MARIA ALMEIDA AMORIM DA COSTA

Bachelor in Computer Science

Adviser: Bernardo Toninho
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chair: Doctor Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

Rapporteur: Doctor Francisco Cipriano da Cunha Martins
Associate Professor, University of the Azores

Adviser: Doctor Bernardo Parente Coutinho Fernandes Toninho
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

Session Kotlin

This document was created using the (pdf/Xe/Lua)LaTeX processor, based on the NOVAthesis template, developed at the Dep. Informática of FCT-NOVA by João M. Lourenço. [43]

Acknowledgements

I would like to thank my adviser, Bernardo Toninho, for his guidance throughout the development of this thesis. His incredible support and patience made for an almost stress-free experience.

Special thanks to my undergrate final project mentor, professor Vítor Duarte, who taught me much about project development and maintenance, and to professor João Lourenço for developing the present thesis template, which no doubt spared me countless hours of work.

To my family, thank you for the constant support and patience; you made this possible.

Abstract

Concurrency and distribution have become essential for building modern applications. However, developing and maintaining these apps is not an easy task. Communication errors are a common source of problems: unexpected messages cause runtime errors, and mutual dependencies lead to deadlocks. To address these issues, developers can define communication protocols that detail the structure and order of the transmitted messages, but maintaining protocol fidelity can be complex if carried out manually. Session types formalize this concept by materializing the communication protocol as a type that can be enforced by the language's type system. In this thesis we present the first embedding of session types in Kotlin: we propose a Domain-Specific Language (DSL) for multiparty session types that lets developers write safe concurrent applications, with built-in validation and integrating code generation in the language's framework.

Keywords: Session types, Concurrency, Kotlin, Domain-specific languages

Resumo

A concorrência e a distribuição têm-se tornado essenciais na construção de aplicações modernas. No entanto, desenvolver e manter estas aplicações não é tarefa fácil. Erros de comunicação são uma fonte comum de problemas: mensagens inesperadas causam erros durante a execução de código, e dependências mútuas levam a *deadlocks*. Para resolver estas questões, é tipico definir protocolos de comunicação que detalham a estrutura e a ordem das mensagens transmitidas, mas garantir o seu cumprimento pode ser complexo se feito manualmente. Os tipos de sessão formalizam este conceito ao materializar o protocolo de comunicação como um tipo que pode ser gerido pelo sistema de tipos da linguagem. Nesta tese apresentamos o primeiro *embedding* de tipos de sessão em Kotlin: propomos uma Linguagem de Domínio Específica para tipos de sessão com múltiplos participantes que permite aos programadores a escrita de aplicações concorrentes seguras, incorporando validação e integrando a geração de código no *framework* da linguagem.

Palavras-chave: Tipos de sessão, Concorrência, Kotlin, Linguagem de Domínio Específico

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Concurrency and distribution have become essential for building modern applications. However, developing and maintaining them is not easy. Programmers have to coordinate processes and define a method of communication, usually either by sharing memory or by sending messages over some channel abstraction. The former is typically easier to make mistakes in: the wrong limitation of critical regions and incorrect usage of synchronization primitives are frequently sources of problems. The latter has the advantage of being more intuitive and can be easily ported from concurrent to distributed environments by changing the transport layer. Unfortunately, simply communicating via channels is not enough to avoid all errors [58]: deadlocks can still occur, and distributed architectures have the additional issue of being susceptible to network errors (lost messages, message reordering). Communication errors between endpoints are one type of error that is common in concurrent applications. They can happen when some endpoint sends a message that the receiver is not expecting or doesn't know how to process, or when endpoints have mutual dependencies, creating a deadlock. It thus becomes critical to develop tools that can mitigate these issues and give developers some safety guarantees.

Inter-process interactions that comprise more than one message are built with some shared agreement (a protocol) in mind that defines the structure and order of the transmitted messages, but maintaining protocol fidelity can be complex if carried out manually by developers. Ensuring all possible paths of the protocol are accounted for and that the correct messages are received and sent at the right moment is an error-prone task - particularly in the presence of iteration. This issue persists for as long as the software is maintained.

Therefore, to help developers build concurrent applications, we need to materialize in some way this shared agreement. This materialization should be responsible for the precise representation of the sequence of interactions (the *session*). Additionally, we need a tool that audits the code and verifies that the protocol is followed. In the best-case scenario, it should statically guarantee the absence of communication errors and deadlocks.

One way of implementing this is to formalize the protocol as a *type* and have the

type system be the tool that enforces its correct usage. This idea is the basis for session types [19], which enforce pre-determined sequences of I/O actions on communication channels. By delegating this responsibility to the type system, we reduce software maintenance costs, eradicate communication errors, and guarantee deadlock-freedom.

However, to implement session types, the language's type system needs to be fairly sophisticated. It needs to track the session type, which evolves as actions on channels take place, potentially in the presence of aliasing of channel references, which requires intricate ownership tracking not typically found in general purpose languages. Typically, special-purpose languages are created to support this from scratch, but these languages are not the ones programmers use in their day-to-day work. Additionally, "mainstream" languages generally do not have type systems that can track the stateful evolution of resources, and this makes it challenging to encode session types.

Domain-Specific Languages are small languages that, as the name suggests, are focused on a particular domain or about solving a specific problem. They can be either *internal* or *external*, as defined by Martin Fowler [15]. Internal (or embedded) DSLs are languages defined within another language, and are typically implemented as a library: they bend the host language in a way that it appears we are programming in another language. External DSLs are not bound to a particular language and usually have custom syntax and parsers. The SQL language and the sed utility [1] are examples of such DSLs. This thesis focuses on the internal kind. Internal DSLs are much more practical to use as they do not require external tools. Our DSL is built on top of the Kotlin language.

Kotlin is a modern, open-source, statically-typed programming language developed by JetBrains that supports both object-oriented and functional programming. It has been the official language for Android development since 2019 [2] and is interoperable with Java: it is possible to call Java code from Kotlin, and Kotlin can be used in Java. It is also multiplatform and compiles to the JVM, Javascript, and native binaries. One can develop an application that targets all three; this is useful, for example, when developing multiplatform libraries or to share common business code between mobile and web applications. Nullability is encoded in the type system, making it null-safe and avoiding runtime errors related to null values.

Kotlin also has first-order functions, lambdas and extension functions (functions that add functionality to existing classes). Putting all this together, we can create semi-declarative type-safe builders to create a DSL inside Kotlin. Type-safe builders are, in essence, the builder pattern with some extra type-safety guarantees: they statically ensure that all mandatory properties are initialized and that methods are used in the right context.

For example, the Gradle build tool comes with a Kotlin DSL for writing build scripts in Kotlin, since version 5.0 [3]. By using a statically typed language instead of Groovy,

[1] https://www.gnu.org/software/sed/manual/sed.html
[2] https://developer.android.com/kotlin/first
[3] https://docs.gradle.org/5.0/release-notes.html#kotlin-dsl-1.0

a dynamically typed language, IDE's can provide useful features like code completion, refactoring, error highlighting, and source code navigation.

Goals In this thesis we propose a Domain-Specific Language (DSL) in Kotlin for multiparty session types with built-in validation that lets developers write safe concurrent applications. Multiparty session types [20, 10] extend the theory behind session types to account for more than two participants. The goal is to provide safety guarantees, such as *communication safety* (no discrepancy between the types of the messages sent and received) and *protocol fidelity* (all messages are accounted for), using a combination of static (compile-time) and dynamic (runtime) verifications and exploring Kotlin's capabilities for DSL development, including type-safe builders and metaprogramming tools.

Contributions

- We present the first embedding of session types in Kotlin;
- We present a Kotlin-based DSL for multiparty session types that integrates type verification (*i.e.* protocol validation) and code generation in the language's framework;
- We provide an implementation that allows for both concurrent and distributed programming, based on coroutines and sockets;
- We provide a parametric endpoint implementation that seamlessly supports both fluent and callback-based styles;
- Our DSL allows type refinements, offering the possibility to restrict message values and the relationships between them.

Outline In Chapter 2 we introduce fundamental concepts to this thesis, such as behavioural types, binary and multiparty session types, and typestates. We discuss existing implementations of session types in other languages, and explore relevant Kotlin features for DSL implementation. Chapter 3 describes and evaluates our contributions, and Chapter 4 discusses future work.

2

Background

When building concurrent applications, there are two main ways of communicating between processes: by sharing memory and by passing messages over some kind of channel. This work focuses on the latter. Most applications that use messages as a mean of communication have some form of structure that the programmer has in its mind when writing the program. For example, if we have a server that receives two integers and return their sum, we can design an informal protocol that looks like this:

Figure 2.1: An adder server and a client

Server

```
1 receive an integer
2 receive an integer
3 send their sum
```

Client

```
1 send an integer
2 send an integer
3 receive their sum
```

A problem that commonly arises is how to guarantee that the protocol is followed by both sides. It's easy to verify with such a small example, but, if there were more messages, or more participants, or if the flow branched depending on some choice, or if the protocol suffers modifications, the burden of checking for errors lies on the programmer. This issue encouraged the research and development of several languages and tools in the realm of type theory, which we overview in the following sections.

2.1 Behavioural Types

Behavioural types arise in the context of process calculi and programming languages as a way of capturing program behaviour at the type level [23]. This typing discipline specifies properties related to concurrent programs such as causality and choice and seek to ensure safety properties such as absence of communication errors, race freedom, and deadlock freedom, in addition to liveness properties such as progress. To this end, the processes' actions are often partially ordered to avoid cyclic dependencies, and the channels are classified to ensure reliability. Combining these two concepts guarantees deadlock-freedom [32].

Consider the following example from [32]: Assume there is a process that receives a value along x ($x?[]$) and then sends a value along y ($y![]$). In the type system proposed by Kobayashi [32], it has the following type judgement:

$$x : \updownarrow []^{t_x}, y : \updownarrow []^{t_y}, (t_x, t_y) \vdash x?[] \cdot y![]$$

where t_i specify time tags, and the relation (t_x, t_y) defines that x may be used before y. In contrast, a process that receives on y and then sends over x has the following type:

$$x : \updownarrow []^{t_x}, y : \updownarrow []^{t_y}, (t_y, t_x) \vdash y![] \cdot x?[]$$

When composed in parallel $x?[] \cdot y![] | y![] \cdot x?[]$, a cyclic ordering is detected through the time tags $\{(t_x, t_y), (t_y, t_x)\}$, since they define that communication over x may be done after y and, at the same time, communication over y may be performed after x.

Kobayashi et al. [33] build on this idea by developing a type reconstruction algorithm, an extension of the one developed by Igarashi and Kobayashi [25], to check processes without type annotations.

Igarashi and Kobayashi [24] propose a generic framework of system types for concurrent programming languages that express types as abstract processes. They show that deadlock-freedom and race-freedom can be derived as instances of this generic type system. The concept of conversation types, more closely related to session types (Section 2.2), is presented by Caires and Vieira [7] as a generalization of binary session types to concurrent multiparty conversations. Additionally, they allow dynamic conversations: participants may join and leave as the conversation progresses.

2.2 Binary Session Types

Session types were first introduced by Honda, Vasconcelos, and Kubo [19], as a method of describing complex interactions between two communicating processes. Their proposal stands on three pillars:

- the *session*, a chain of interactions between two participants;
- Three basic communication primitives:
  - *value passing*;
  - *labelled branching*: a purified form of method invocation, where one side is expected to make a choice;
  - *delegation*: the ability to pass a channel to another process.
- A basic type discipline for the communication primitives, to ensure two communicating processes have compatible patterns.

The compiler, having access to the *session type*, can verify that the operations are performed in an order that everyone expects, ensuring communication safety (no discrepancy between the types of the messages sent and received), session fidelity (all messages are accounted for in the protocol), and, if there is no session interleaving (more than one session executing concurrently), deadlock-freedom. Session types can also be described as a special case of behavioural type: they present a stricter type language, allowing fewer valid programs, but have the advantage of being easier to develop and implement in programming languages.

Figure 2.2: Session types

$$
\begin{array}{rr}
S ::= \text{bool} \mid \text{nat} \mid \ldots & \text{Sorts} \\
U ::= S \mid T & \text{Exchange types} \\
 & \\
T ::= !U.T & \text{Send} \\
\mid ?U.T & \text{Receive} \\
\mid \&\{l_i : T_i\}_{i \in I} & \text{Branch} \\
\mid \oplus\{l_i : T_i\}_{i \in I} & \text{Select} \\
\mid \mu t.T \mid t & \text{Recursion} \\
\mid end & \text{Termination}
\end{array}
$$

With respect to the syntax, depicted in Figure 2.2, we use S to represent basic types like booleans and numbers, T to range over sessions types, and U to represent exchange types, which can be S or T. $!U.T$ and $?U.T$ are used to *send* and *receive* values or channels (session delegation), continuing with T. The *branch* type $\&\{l_i : T_i\}_{i \in I}$ denotes an *external* choice, and the *select* type $\oplus\{l_i : T_i\}_{i \in I}$ represents an *internal* choice. The labels l_i range over an index set I. Recursion is modelled by the types $\mu t.T$ and t, and sessions terminate with *end*.

Two endpoints of a session are compatible if their types are dual: every *send* must be matched with *receive*, every *internal* choice must be matched to an *external* choice, and vice-versa. Figure 2.3 lists these duality rules in detail.

Now that we have defined some grammar rules for session types, we can specify session types for the Adder example from Figure 2.1: the server must receive two values and send one, $?Int.?Int.!Int.end$, and the client must do the opposite: $!Int.!Int.?Int.end$.

In Figure 2.4 we have an example of an arithmetic server that is capable of performing addition and subtraction, offering a choice (&) between `Add` and `Sub`. The client must have the dual session type: an *internal* choice ($\oplus$) between the same labels (`Add`, `Sub`). The type of each session for each label need to be dual as well: $?Int$ becomes $!Int$ and vice-versa, following the rules defined in Figure 2.3.

Figure 2.3: Duality of session types

$$
\begin{array}{rcl}
\overline{!S.T} & \triangleq & ?S.\overline{T} \\
\overline{?S.T} & \triangleq & !T.\overline{T} \\
\overline{\&\{l_i : T_i\}_{i\in I}} & \triangleq & \oplus\{l_i : \overline{T_i}\}_{i\in I} \\
\overline{\oplus\{l_i : T_i\}_{i\in I}} & \triangleq & \&\{l_i : \overline{T_i}\}_{i\in I} \\
\overline{\mu t.T} & \triangleq & \mu t.\overline{T} \\
\overline{t} & \triangleq & t \\
\overline{end} & \triangleq & end
\end{array}
$$

Figure 2.4: An arithmetic server and a client

Server

```
1 Server = &{
2   Add: ?Int.?Int.!Int.end,
3   Sub: ?Int.?Int.!Int.end
4 }
```

Client

```
1 Client = ⊕{
2   Add: !Int.!Int.?Int.end,
3   Sub: !Int.!Int.?Int.end
4 }
```

If the types for both ends are dual, and we can prove session fidelity (*i.e.*, the code follows the protocol), then communication safety is guaranteed: every send has a matching receive and every message is expected despite the fact that we are using the same channel for messages of different types. Processes that follow these types do not get "stuck": they are deadlock-free.

Figure 2.5: Interleaving of binary sessions

Process A

```
1 receive from B (s1)
2 send to C (s2)
```

Process B

```
1 receive from C (s3)
2 send to A (s1)
```

Process C

```
1 receive from A (s2)
2 send to B (s3)
```

These properties are only guaranteed for processes that act on a single session: if we introduce session interleaving, deadlock-freedom is lost. Figure 2.5 shows an interleaving of three binary sessions: *s1*, *s2* and *s3*. In these sessions, an integer is exchanged. Process A shares session *s1* with B and session *s2* with C; session *s3* is between B and C. There are no type errors but the processes are blocked waiting for a message that will never arrive: they are deadlocked. When we need to describe interactions between more than two participants, binary sessions are not enough to guarantee deadlock-freedom.

2.3 Multiparty Session Types

Multiparty session types [20, 10] extend the theory behind binary session types to account for more than two processes. Honda et al. [20] introduces a new kind of type in which interactions involving multiple actors are abstracted as a global scenario: the global type. The global type is a shared agreement among the peers and can be used to *project*

local sessions which can be used to typecheck individual peers. Projection is defined inductively on the global type Figure 2.7.

Figure 2.6: Global session types

$$
\begin{array}{rcll}
G & ::= & p \to q : \langle U \rangle . G & \text{Exchange} \\
 & | & p \to q : \{l_i : G_i\}_{i \in I} & \text{Branching} \\
 & | & \mu t . G \mid t & \text{Recursion} \\
 & | & end & \text{Termination}
\end{array}
$$

We define the type for the global session in Figure 2.6. We use p, q to represent peers and U has the same meaning as in Figure 2.2. The exchange type $p \to q : \langle U \rangle . G$ defines that peer p sends either a value or a channel to peer q, and proceeds with G ($p \neq q$); the branching type $p \to q : \{l_i : G_i\}_{i \in I} . G$ means that peer p sends a label l_i to peer q ($p \neq q$) and then continues with G_i.

Figure 2.7: Projection of G onto participant q ($G \upharpoonright q$)

$$
(p_1 \to p_2 : \langle U \rangle . G) \upharpoonright q = \begin{cases} !\langle U, p_2 \rangle . (G \upharpoonright q), & \text{if } q = p_1 \\ ?\langle U, p_1 \rangle . (G \upharpoonright q), & \text{if } q = p_2 \\ G \upharpoonright q & \text{otherwise} \end{cases}
$$

$$
(p_1 \to p_2 : \{l_i : G_i\}_{i \in I}) \upharpoonright q = \begin{cases} \oplus \langle p_2, \{l_i : G_i \upharpoonright q\}_{i \in I} \rangle, & \text{if } q = p_1 \\ \& \langle p_1, \{l_i : G_i \upharpoonright q\}_{i \in I} \rangle, & \text{if } q = p_2 \\ G_k \upharpoonright q, & \text{where } k \in I \wedge q \neq p_1 \wedge \\ & q \neq p_2 \wedge G_i \upharpoonright q = G_j \upharpoonright q \; \forall i, j \in I \end{cases}
$$

$$
(\mu t . G) \upharpoonright q = \begin{cases} \mu t . (G \upharpoonright q), & \text{if } (G \upharpoonright q) \neq t \\ end, & \text{otherwise} \end{cases}
$$

$$
t \upharpoonright q = t
$$

$$
end \upharpoonright q = end
$$

The projection function for global types is defined in Figure 2.7. The first rule specifies the projection of the exchange type. If q, the role we are projecting to, is the sender, then it needs to send a message to the other role: $!\langle U, p2 \rangle$; if q is receiving, it needs to receive the message: $?\langle U, p1 \rangle$. If q does not participate in the exchange, ignore the exchange. In all cases, we must continue projecting the rest of the type.

The second rule refers to the projection of the branching type. If q is the one making the decision, it is projected as an internal choice ($\oplus$); if it is the one offering a decision, it is projected as an external choice ($\&$). In both cases, we must continue projecting the branches. If q is not making or offering the decision, and *all* branches when projected onto q are equal,then the local type is the projection of any of them.

This implies that, when projecting a branching type and q is not choosing or offering a choice, and if the projection $G_k \upharpoonright q$ is not equal for all branches, the function is undefined. If a role does not know the outcome of a choice, its local type needs to be the same for every label. Furthermore, recursion needs to be guarded. The last rule defines the projection of termination, which is always *end*.

A global type is *well-formed* if the projection $G \upharpoonright q$ is defined for all participants. Figure 2.8 details a malformed global type G_1. This type describes a protocol in which, depending on the choice a makes, b either sends a message to c or to a. The local types that correspond to the projections for a and b are shown in Figure 2.9. The local type for c is undefined because, as c does not have knowledge of the choice a made, does not know whether it should receive a message from b or not. Formally, $G_1 \upharpoonright c$ is undefined because $(b \to c : \langle U \rangle.end) \upharpoonright c = ?\langle U, b \rangle.end$ is not equal to $(b \to a : \langle U \rangle.end) \upharpoonright c = end$. This global type, according to the projection function defined in Figure 2.7, is therefore malformed. This clause ensures that participants that are not involved in a choice behave the same, regardless of the outcome of the choice.

Figure 2.8: Malformed global type G_1

$$
\begin{aligned}
G_1 \triangleq\ & a \to b : \{ \\
& \quad L_1 : b \to c : \langle U \rangle.end, \\
& \quad L_2 : b \to a : \langle U \rangle.end \\
& \}
\end{aligned}
$$

Figure 2.9: Local projections of G_1

$$
\begin{aligned}
& G_1 \upharpoonright a = \oplus\langle b, \{ \\
& \qquad L_1 : end, \\
& \qquad L_2 : ?\langle U, b \rangle.end \\
& \quad \}\rangle \\
& G_1 \upharpoonright b = \&\langle a, \{ \\
& \qquad L_1 : !\langle U, c \rangle.end, \\
& \qquad L_2 : !\langle U, a \rangle.end \\
& \quad \}\rangle \\
& G_1 \upharpoonright c = \text{undefined}
\end{aligned}
$$

There is a different definition for the branching projection that takes a more relaxed approach when projecting a role that does participate in a choice and has non-identical branches. Carbone et al. [8] introduced the notion of *mergeability* and defined the projection of the branching as a merge of the projections of the branches. Using this operator allows for more global types to be projectable.

Taking the global type G_2 defined in Figure 2.10 as an example, its projection onto c ($G_2 \upharpoonright c$) is only defined using this notion of mergeability. Although projecting the L_i branches onto c result in a different type, we can merge them and have c offer both K_i branches to b, regardless of a's decision.

Figure 2.12 describes the Two Buyer protocol [20], in which the two buyers, A and B, combine their money to purchase an expensive item from $Seller$. Buyer A starts by sending a message to $Seller$, with the name of the item they intend to buy. Afterwards,

Figure 2.10: Global type G_2

$$
\begin{aligned}
G_2 \triangleq\ & a \to b : \{ \\
& \quad L_1 : b \to c : \{K_1 : end\}, \\
& \quad L_2 : b \to c : \{K_2 : end\} \\
& \}
\end{aligned}
$$

Figure 2.11: Local projections of G_2

$$
\begin{aligned}
G_2 \upharpoonright a = \oplus\langle b, \{ & \\
& L_1 : end, \\
& L_2 : end \\
\}\rangle & \\
G_2 \upharpoonright b = \&\langle a, \{ & \\
& L_1 : \oplus\langle c, \{K_1 : end\}\rangle, \\
& L_2 : \oplus\langle c, \{K_2 : end\}\rangle \\
\}\rangle & \\
G_2 \upharpoonright c = \&\langle b, \{ & \\
& K_1 : end, \\
& K_2 : end \\
\}\rangle &
\end{aligned}
$$

Seller sends the quote to both buyers. With this information, A sends to B the amount of money it can give, and B decides whether it is enough by choosing between two branches: Ok, sending the address to the seller and receiving the date of delivery, or $Quit$, terminating the protocol. Figure 2.13 defines corresponding the global type. The local types that correspond to applying the projection function (Figure 2.7) to the global type onto each participant are shown in Figure 2.14.

Figure 2.12: The Two Buyer protocol

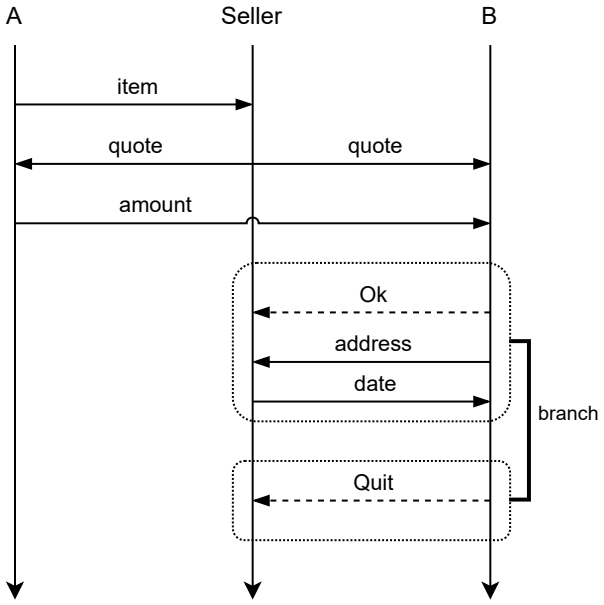


Figure 2.13: Global type $TwoBuyer$

$$
\begin{aligned}
TwoBuyer \triangleq\ & \\
& a \to s : \langle String\rangle. \\
& s \to a : \langle Int\rangle. \\
& s \to b : \langle Int\rangle. \\
& a \to b : \langle Int\rangle. \\
& b \to s : \{ \\
& Ok : \\
& \quad b \to s : \langle String\rangle. \\
& \quad s \to b : \langle String\rangle. \\
& \quad end, \\
& Quit : \\
& \quad end \\
& \}
\end{aligned}
$$

If we define a well-formed global type for the interleaving of binary sessions detailed

Figure 2.14: Local projections of $TwoBuyer$

$$
\begin{aligned}
TwoBuyer \upharpoonright a \triangleq\ & !\langle String, s\rangle.?\langle Int, s\rangle. \\
& !\langle Int, b\rangle.end
\end{aligned}
$$

$$
\begin{aligned}
TwoBuyer \upharpoonright b \triangleq\ & ?\langle Int, s\rangle.?\langle Int, a\rangle. \\
& \oplus\langle s, \{ \\
& \quad Ok : !\langle String, s\rangle. \\
& \qquad ?\langle String, s\rangle.end, \\
& \quad Quit : end \\
& \}\rangle
\end{aligned}
$$

$$
\begin{aligned}
TwoBuyer \upharpoonright s \triangleq\ & ?\langle String, a\rangle. \\
& !\langle Int, a\rangle.!\langle Int, b\rangle. \\
& \&\langle b, \{ \\
& \quad Ok : ?\langle String, b\rangle. \\
& \qquad !\langle String, b\rangle. \\
& \qquad end, \\
& \quad Quit : end \\
& \}\rangle
\end{aligned}
$$

in Figure 2.5, we'd have no more deadlocks. Since the protocol would start with an exchange, some process would need to send the first message before receiving any. Figure 2.15 shows a well-formed global type that describes the interaction. In this example, c sends the first message, and no processes will ever become deadlocked.

Figure 2.15: Global type G_3

$$
\begin{aligned}
G_2 \triangleq\ & c \to b : \langle Int\rangle. \\
& b \to a : \langle Int\rangle. \\
& a \to c : \langle Int\rangle. \\
& end
\end{aligned}
$$

2.4 Refinements

Session types are useful to describe communication *structure*, but they are unable to capture information about the communicated data. Take, for example, the protocol described in Figure 2.4. When the client chooses the `Add` branch and sends two integers, it expects a value equal to their sum, but there is no guarantee that the server's answer is correct. In other words, session types represent the *choreography* but cannot describe message content apart from its type.

The general concept of refinement types can be applied to multiparty session types [6, 57, 60] in order to express properties and constraints on communicated values and their relationships. In this extended setting, messages are labeled so that it is possible to refer to them in predicates. Figure 2.16 illustrates a refined global protocol. In this example, the client messages are labeled as `a` and `b`, while the server response, `c`, is restricted to be equal to the sum of the previous messages.

Figure 2.16: Refined addition

$$
\begin{aligned}
Adder \triangleq\ & \\
& client \to server : \langle Int \rangle [a]. \\
& client \to server : \langle Int \rangle [b]. \\
& server \to client : \langle Int \rangle [c, a + b = c]. \\
& end
\end{aligned}
$$

To implement such an extension, additional validations are required. The first challenge is related to the local vision that each participant has of the global interaction: type projection needs to guarantee that endpoints have all the information they need to verify the refinements. Secondly, the refined interactions needs to be *satisfiable*, which means that there should be some combination of message values that allow the predicates to be true. To perform these validations, SMT solvers are often used [45, 60, 13] in combination with runtime assertions [60] or proof objects [57].

2.5 Typestates

Typestates, a kind of behavioural type, were originally introduced by Strom and Yemini [53] as a way of restricting the operations available on an object to a subset that depends on its state.

Take, for example, an object representing a file with four operations: open, read, write, and close. Intuitively, this object can have two states: the first, which is the initial state, should only have the open operation available. The object progresses to the next state with the open operation, in which it can be read, written to, or closed. In this state, closing the file changes it back to the initial state, making it impossible to read or write before opening it again. The idea is having the type system enforce this pattern to statically guarantee that it is impossible to read and write from a closed file.

Arguably, session types could be perceived as typestates defined on channel objects with send and receive operations: they can both be seen as a finite state machine, but session types are typically focused on concurrency. Concretely, they are more closely related to session endpoints that follow the fluent API pattern: the class chaining via method calls have a similar look to how the object is manipulated. A popular tool in the session types ecosystem is Scribble [59], which defines a language for describing multiparty session types. Its toolchain is commonly extended to generate APIs for general-purpose languages. StMungo [38] bridges typestates and session types by generating typestates definitions from multiparty communication protocols written in the Scribble language.

2.6 Implementations of Session Types

Most presentations of session types are developed in special-purpose languages instead of general-purpose languages, and are typically focused on providing as much safety guarantees as possible rather than user experience. Additionally, not all languages support direct implementations of session types: the fundamental challenge comes from the type system's need to track resources and channel types, potentially in the presence of aliasing. In this section, we explore two ways of approaching this subject: natively or through embedding.

2.6.1 Natively session Typed Languages

Native implementations come in two categories: included in languages created to support them (usually minimalistic, with limited usability), or as an extension of a preexisting language. In both cases, they implement the necessary sophisticated types to represent type evolution.

Toninho et al. [56] presents a functional language that integrates a Curry-Howard interpretation of linear sequent calculus as session typed processes. Processes are encapsulated in a *contextual monad* that also contain all the channels. Griffith [17] explores logically motivated session types and polarization, and presents a language called SILL. Its interpreter is written in Ocaml.

Das and Pfenning [13] present Rast (Resource-Aware Session Types), a language based on binary session types governing the interaction of two processes along a single channel. Supports arithmetic type refinements as well as ergometric and temporal types to measure the total work and span of the programs. It is implemented in Standard ML and the algorithm for type equality is presented in their following work [14].

FreeST [2] is an implementation of the context-free session type language introduced by Thiemann and Vasconcelos. Context-free session types allow left recursion instead of the usual tail-recursion, enabling the transmission of tree-structured data in a type-safe way [55].

Fowler et al. [16] extend Links, a functional web programming language, with support for session types with failure handling using the existing effect handlers of the language. They introduce three new terms to support failure handling: *cancel*, to explicitly cancel a session endpoint, *raise*, to raise an exception, and a *try-catch* term to handle failures. These terms map onto existing Links constructs. Effect handlers are a generalization of exception handlers.

Additionally, there are some implementations that extend existing languages. Hu, Yoshida and Honda [22] extend Java with session types, with support for delegation subtyping. This strategy comes with the cost of locking the developer to a specific Java version (in this case, 1.4).

All the works mentioned above have completely static guarantees of error-free communication and, except for the last one, deadlock-freedom.

Mungo [44] is a tool that can be used to statically check the order of method calls in Java. An annotation is added to classes, associating them to a protocol that defines the sequence of method calls. The protocol files are generated by StMungo [38, 44], a tool that translates local protocols from Scribble to typestate specifications that Mungo can use.

2.6.2 Embedding Session Types

The second type of implementation is with an embedding in a general-purpose language. Incorporating session types in a "mainstream" language can be challenging: the type system needs to be sophisticated to support the necessary features session typing requires: channel linearity (use exactly once), branch exhaustion, and type duality.

Within this group, we have two approaches: those that aim to provide strong, fully-static guarantees, and those that aim to promote usability by relaxing some verifications to runtime (usually, linearity).

2.6.2.1 Statically checked approach

Fully statically checked implementations are generally very safe, but can be challenging to use for programmers. The host language's type system needs to be flexible, especially to enforce linearity: functional languages are generally good candidates for this approach. Everything is encoded statically and does not have any runtime overhead, as they statically encode everything in the type system. On the downside, error messages may be hard to interpret.

A fully static implementation of session types in OCaml is provided by Imai et al. [26]. It uses a parameterized monad to statically encode multiple simultaneous sessions, and lenses to manipulate a symbol table of the monad.

In Haskell, Pucella and Tov [46] propose a library that handles multiple communication channels typed independently and infers session types automatically. Aliasing is avoided by threading session type information linearly through the system, using an indexed monad. Type classes are used to express the duality of session types. Lindley et al. [41] presents an embedding of GV (a core session-typed functional language, built on Wadler's work on functional calculus) and two implementations of that embedding: one based on the concurrent primitives in Haskell's IO monad and another that expresses concurrency using continuations. Finally, Kokke and Dardha [35] show a deadlock-free implementation of session types, even if the process structure has cycles, using priorities. These priorities are an extension of session types with partial ordering, allowing programs that have cyclic process structure but have an acyclic communication graph.

In Rust, Chen, Balzer and Toninho [9] allow shared session types that support safe aliasing of channels. The channels must be used in mutual exclusion: clients need to

acquire the linear channel to the component, becoming its unique owner, and release it when done.

2.6.2.2 Hybrid approach

Hybrid implementations delegate some checks to runtime. This can be useful because not all languages have sophisticated enough type systems to effectively encode, for example, linear use of resources.

There are several implementations of session types that take this approach. Hu and Yoshida [21] present `scribble-java`, an implementation in Java for multiparty session types based on API generation as an extension of the Scribble protocol language. Linear usage of channels is verified at runtime. Each protocol state is materialized as a distinct channel type that permits only the exact I/O operations according to the protocol. These channels are linked as a call-chaining API that returns a new instance of the successor state for the action performed. The global protocol is defined in Scribble.

Scribble [59] is a language that describes multiparty protocols for communication. It is frequently used to represent and validate global protocols, generate local types, and generate local APIs for other languages, like Java and Scala. By delegating these tasks to Scribble, programmers can avoid the challenges of representing session types and implementing validation.

Listing 2.1 shows an implementation of the Seller endpoint of the Two Buyer protocol (Subsection 2.3) using `scribble-java` [21]. Line 1 instantiates an endpoint for the role *S* inside a *try-with-resources* statement, to automatically close the session. Lines 4 and 5 accept connections from the two buyers, *A* and *B*. Afterwards, we start the protocol by creating an object, for the initial state, `TwoBuyer_S_1`. Starting with it, we execute the protocol using the call-chaining API until it terminates. A *switch* statement is used to branch based on the decision of *B*.

Scalas and Yoshida [47] present a library, `lchannels`, that offers an API for binary session programming in Scala with continuation-passing style programming. Linearity is enforced during runtime. Scalas et al. [48] build on this work by encoding deadlock-free multiparty sessions as a composition of binary sessions. They also extend Scribble.

Neykova et al. [45] propose a library for the specification and implementation of multiparty distributed protocols in F#. It is implemented by extending and integrating Scribble with an SMT solver into the type providers framework. Type providers are a .NET feature for a form of compile-time metaprogramming, designed to bridge between statically typed languages and information spaces (structured data sources like SQL databases or XML data). A type provider works as a compiler plugin that performs on-demand generation of types: it takes a schema, in this case, a Scribble protocol, and generates protocol and role-specific types of an API for implementing the endpoint, with methods for chaining I/O actions. This paper also implements refinements, which are logical constraints over the data. Linearity is enforced at runtime. Refinements that

Listing 2.1: Seller endpoint of the Two Buyer protocol with scribble-java

```
try (MPSTEndpoint<TwoBuyer, S> endpoint = new MPSTEndpoint<>(tb, S.S, new
↪  ObjectStreamFormatter())) {
    Buf<String> buffer = new Buf<>();

    endpoint.accept(new SocketChannelServer(9997), A.A);
    endpoint.accept(new SocketChannelServer(9998), B.B);

    TwoBuyer_S_1 s1 = new TwoBuyer_S_1(endpoint);
    TwoBuyer_S_2 s2 = s1.receive(A.A, title.title, buffer);

    int quoteValue = 100;
    TwoBuyer_S_4 s4 = s2
      .send(A.A, quote.quote, quoteValue)
      .send(B.B, quote.quote, quoteValue);

    TwoBuyer_S_4_Cases cases = s4.branch(B.B);

    switch (cases.getOp()) {
      case ok:
        cases
            .receive(B.B, ok.ok, buffer)
            .send(B.B, EMPTY_OP.EMPTY_OP, Date.from(Instant.now()));
        break;

      case quit:
        cases.receive(B.B, quit.quit);
        break;
    }
}
```

cannot be verified statically are enforced with assertions during runtime.

In Rust, Lagaillardie et al. [40] present an implementation of multiparty session types as a wrapper of the library for binary session types done by Kokke and Wen [34]. Local Rust types can be generated by Scribble, guaranteeing deadlock-freedom (because they were projected from a well-formed global type), or written by the programmer and statically checked to ensure reception error safety. Rust has some features that help guarantee linear use of channels: its affine type system guarantees at-least-once usage of variables, and, to prevent dropped sessions, `[#must_use]` is employed to annotate the definitions of the *send*, *receive* and *end* operations: this causes Rust to emit a warning whenever a session is dropped. The Rust compiler statically guarantees that more-than-once usage never happens.

2.6.3 Session Types as DSLs

The original theory behind multiparty session types seek to provide static guarantees such as deadlock-freedom, communication safety, and protocol fidelity. But, in practice, mapping these concepts to mainstream languages is a difficult task: very few languages have rich type systems capable of tracking resources and channel types, potentially in

the presence of aliasing. Nonetheless, there has been extensive work on static implementations for the languages that do support it, such as in OCaml [26], Haskell [46, 41, 35], and Rust [29, 34, 9].

In the context of multiparty session types, there is an implementation in Rust, `mpst-rust` [40], that uses Scribble to generate local Rust types. Additionally, Cutner et al. present `rumpsteak` [11, 12]. They use νScr [1], a toolkit that manipulates Scribble protocols, for the same effect. In the OCaml ecosystem, Imai et al. present `ocaml-mpst` [27], a library that uses global combinators as a way of representing global types.

However, strong static guarantees often come at the cost of usability. The implementations typically result in less idiomatic and user-friendly APIs, either requiring users to describe protocols in an external environment (Scribble, νScr) or in a verbose embedded DSL. `priority-sesh` [35] tries to sidestep this problem by encoding linearity using the Linear Haskell language extension [4] instead of implementing it in Haskell's type system. In an effort to improve compatibility between linear code and the standard library, `linear-base` [42] was created. It contains linear variants of common data types and classes, as well as some useful abstractions. The work of Zhou et al. [60] is a special case: its callback-based approach allows them to dispense linearity checking completely with inversion of control.

In Listing 2.2, we show an example of a type error when misusing types with Rusty Variation, now called `sesh` [34, 49]. We declare a simple protocol type from the perspective of the server in Line 6: it receives two integers and returns their sum. The dual type, for the client, is implicitly derived by the `fork` call. In this example, the client does not send the second integer (note that Line 16 is commented out). The type error, as the compiler kindly reports, is in Line 17: session `s` should have type `Recv` instead of `Send`. In other words, the session should be used to receive a value, not sending one. Two aspects can be highlighted: type definitions can become quite convoluted in bigger protocols and undermine code readability and maintainability. Error message readability is also important: type errors can be hard to interpret, although Rust's compiler produces friendlier error messages than most.

On the other hand, we have implementations that take a more pragmatic approach, either forced by limitations of the language (*e.g.*dynamically typed languages) or as a deliberate choice. In an attempt to provide a better user experience and more idiomatic APIs, some guarantees are relaxed: typically, this includes linearity checks. Code generation is quite common as well. There are implementations in Java [21], Scala [48], and F# [45].

All of these have similar structure: the global protocol definition is specified in the Scribble language (which effectively is an external DSL), that corresponds to the session type definition in Line 6 of the previous example. The global protocol is validated, projected to local protocols, and finite state machines are created for each participant. A class in the target language is generated for each state, containing methods that correspond to the transitions between them. These methods perform the necessary I/O operations that

Listing 2.2: Error example from `sesh` (Rust)

```rust
extern crate sesh;

use std::error::Error;
use sesh::*;

type AddServer = Recv<i64, Recv<i64, Send<i64, End>>>;

fn adder() -> Result<(), Box<dyn Error>> {
    let s = fork(move s: AddServer {
        let (i, s) = recv(s)?;
        let (j, s) = recv(s)?;
        let s = send(i + j, s);
        close(s)
    });
    let s = send(10, s);
    // let s = send(12, s);
    let (r, s) = recv(s)?;
    println!("{0}", r);
    close(s)
}
fn main() {
    adder();
}
```

```
error[E0308]: mismatched types
   --> src/main.rs:17:23
    |
17  |     let (r, s) = recv(s)?;
    |                  ---- ^ expected struct `sesh::Recv`, found struct `sesh::Send`
    |                  |
    |                  arguments to this function are incorrect
    |
    = note: expected struct `sesh::Recv<_, _>`
               found struct `sesh::Send<i64, sesh::Recv<i64, sesh::End>>`
```

were previously validated and return an instance of next state class. Linearity is enforced at runtime, which means that we lose static guarantees of deadlock-freedom.

Our DSL is different in the sense that it is *internal*: the features presented in Section 2.7 allow us to create a library that seamlessly blends with the enclosing code. Furthermore, by offering two options for endpoint code (fluent API and callback-based), developers can choose between the more natural chained-call style with runtime verifications or the linear-by-construction design.

2.7 Kotlin

Kotlin is a modern, open-source, null-safe, statically-typed programming language that supports both object-oriented and functional programming. It is multiplatform, and can target the JVM, JavaScript, and native code.

Notably, Kotlin provides some features that streamline the implementation of a DSL,

which we overview in the following section.

2.7.1 Functions and Lambdas

Kotlin treats functions as first-class citizens: they can be stored and passed around as arguments and returned from other functions. The `map` function operates on a list, applies the supplied function to each element, and returns the modified list. For example, if we wanted to double all the elements of a list of integers, we could use `map` and pass it the lambda `{n -> n * 2}`. Listing 2.3 shows all the different ways we can invoke `map` with a lambda: Line 4 corresponds to the basic method call everyone is accustomed to, Line 5 shows that we can move the lambda out of the parentheses if it is the last argument, and Line 6 shows that when the lambda is the only argument, we can omit the parentheses entirely. We can also omit the argument definition in the lambda and use `it` to refer to the argument, as shown in Line 7 .

Listing 2.3: Lambdas as arguments in Kotlin

```
1 val numbers = listOf(0, 1, 2)
2
3 // All equivalent: [0, 2, 4]
4 numbers.map({ n -> n * 2})
5 numbers.map() { n -> n * 2}
6 numbers.map { n -> n * 2}
7 numbers.map { it * 2 }
```

It is also possible to extend a class with new functionality without inheritance, using **extension functions**. To declare an extension function, we write the *receiver type* before the function name. Listing 2.4 declares an extension function `double` that adds functionality to lists of integers, which are the *receiver type*. Inside the body of the function, `this` refers to the *receiver object*; that is why the body of the `double` function is simply invoking `map` on the receiver (the list), passing as an argument a lambda that multiplies each element by two. The `this` expression can even be omitted, as it implicitly refers to the receiver object.

Listing 2.4: An extension function

```
1 fun List<Int>.double() = this.map {it * 2}
```

Function types with receiver can be instantiated with a special form of function literals[1]: **function literals with receiver**. Inside the function literal, `this` refers to the receiver object, like extension functions.

Listing 2.5 declares a function literal with receiver that has the same functionality as the extension function declared in Listing 2.4.

[1] Function literals are functions that are not declared but passed as an expression.

Listing 2.5: Function literal with receiver

```
1 val double: List<Int>.() -> List<Int> = { map { it * 2 } }
```

2.7.2 Type-safe builders

Kotlin has support for type-safe, statically typed builders that allow us to create DSLs in a semi-declarative way. This is achieved by combining functions as builders, functions with receivers and the fact that in Kotlin we can move the last argument of a function outside the parentheses if it is a lambda and even omit the parentheses if there are no other arguments, as shown previously (Subsection 2.7.1).

Figure 2.17 shows how we can use type-safe builders to write HTML code in a more idiomatic way in Kotlin. We start by calling the `html` function, passing a lambda as the argument. In the lambda's body, we call the `head` and `body` methods of the `HTML` class (the lambda receiver), once again passing lambdas as arguments. The unary plus (`+`) method of the String class is used to add a text element to the children of the `this` element. Line 11 shows how we can pass a mandatory argument to create a link element (`href`). Lines 18-19 generate two list items, showing that we can write any code we want inside the lambdas.

Figure 2.17: Kotlin DSL for HTML (Simplified example from the kotlin docs)

HTML definition

```
1  html {
2    head {
3      title { +"HTML encoding with Kotlin" }
4    }
5    body {
6      h1 { +"HTML encoding with Kotlin" }
7      p {
8        +"an alternative markup to HTML"
9      }
10
11     a(href = "http://kotlinlang.org") {
12       +"Kotlin"
13     }
14
15     p {
16       +"some text"
17       ul {
18         for (i in 1..2)
19         li { +"${i}*2 = ${i*2}" }
20       }
21     }
22   }
23 }
```

Generated HTML

```
1  <html>
2    <head>
3      <title>
4        HTML encoding with Kotlin
5      </title>
6    </head>
7    <body>
8      <h1>
9        HTML encoding with Kotlin
10     </h1>
11     <p>
12       an alternative markup to HTML
13     </p>
14     <a href="http://kotlinlang.org">
15       Kotlin
16     </a>
17     <p>
18       some text
19       <ul>
20         <li>
21           1*2 = 2
22         </li>
23         <li>
24           2*2 = 4
25         </li>
26       </ul>
27     </p>
28   </body>
29 </html>
```

2.7.3 Metaprogramming

KotlinPoet [37] is a Kotlin and Java API for generating Kotlin source files. It has builders, method chaining, and models for Kotlin files, classes, interfaces, objects, type aliases, properties, functions, constructors, parameters, and annotations. This makes generating code easier and makes it is less error-prone than simply writing plain text to a file.

In Figure 2.18 we show how we can generate a `Person` class. Using kotlinpoet's `FileSpec` builder, we start by adding a comment (Line 2). We then, in Lines 3-10, define the class. Lines 4-6 define the primary constructor with an argument `name`, Lines 7-9 define a property `name` that is initialized with the value of the argument.

Figure 2.18: Generating a class with kotlinpoet

```
1  val file = FileSpec.builder("", "Person")
2    .addComment("Generated file")
3    .addType(TypeSpec.classBuilder("Person")
4      .primaryConstructor(FunSpec.constructorBuilder()
5        .addParameter("name", String::class)
6        .build())
7      .addProperty(PropertySpec.builder("name", String::class)
8        .initializer("name")
9        .build())
10     .build())
11 .build()
12
13 file.writeTo(System.out)
```

Output

```
// Generated file
import kotlin.String

public class Person(
  public val name: String
)
```

2.7.4 Coroutines

Coroutines are Kotlin's lightweight threads. They follow the principle of structured concurrency and can only be launched inside a *coroutine scope* that limits their lifetime.

Figure 2.19 shows how we can build programs with them. In Line 2, we use `runBlocking`, a coroutine builder. It blocks the current thread until the coroutine passed as an argument completes. We need to use this function to bridge *blocking* code with *suspending* style code. *Suspend* functions are special functions that can be paused and resumed and can only be used inside other suspend functions. `Channel`'s `send` and `receive` methods are examples of suspending functions. The last argument of the `runBlocking` function is the coroutine (and also a lambda with receiver).

The `launch` function is also a coroutine builder, but this one does not block the current thread. Both coroutines run concurrently - the one started in the call to `runBlocking` and the one initiated by `launch`.

2.7.5 Channels

Kotlin implements channels as a way of sending messages between coroutines. Channels are parameterized with the type of message they transmit, behave like a queue, and can have a buffer. Sending may *suspend* execution if the buffer is full. The same can happen

Figure 2.19: Coroutines and channels

```
1  fun main() {
2    runBlocking {
3      val channel = Channel<Int>()
4
5      launch {
6        for (x in 1..3) channel.send(x)
7      }
8      repeat(3) { println(channel.receive()) }
9      println("Done!")
10   }
11 }
```

Output

```
1
2
3
Done!
```

when receiving if the buffer is empty. It is possible to define a channel with an "unlimited" buffer: With these, sending never suspends and the buffer may grow infinitely until an out-of-memory exception occurs.

Sending and receiving are also *fair*, in the sense that they respect the order of invocation: the first coroutine that invokes *receive* is the first one to receive a value and resume execution.

Figure 2.19 illustrates how can we use channels to communicate between coroutines. Line 3 declares a new unbuffered `Channel` that transmits integers. In the coroutine declared inside `launch`, we send three messages; at the same time, the main coroutine receives them, on Line 8.

3

Developed Work

We present an idiomatic implementation of Multiparty Session Types in Kotlin that explores the features of the language. We provide a DSL, built upon the features mentioned in Section 2.7, that enables programmers to declare and use session types in a practical way [50].

Figure 3.1: Overview of the library

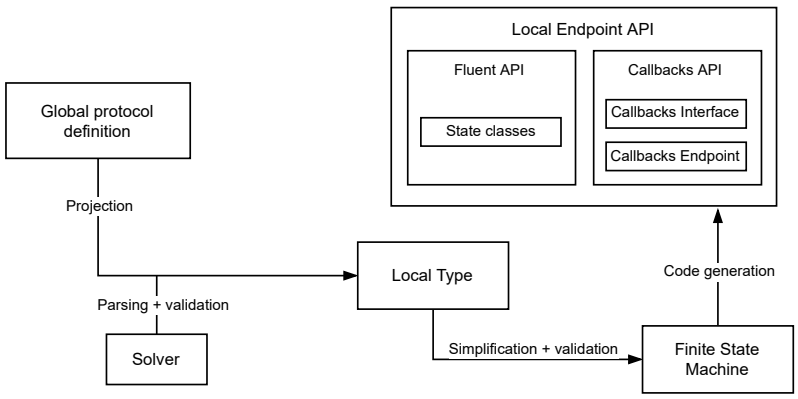


Figure 3.1 provides a high-level view of the library. The global protocol, written by the user, is converted into a recursive global type. If the global type uses refinements, they are parsed, and a solver is invoked to guarantee satisfiability. Afterward, a projection function is applied to the global type, and recursive local types are generated. Finite state machines are created from local types and then used to generate local endpoint APIs. Table 3.1 provides additional implementation statistics.

Table 3.1: Project statistics

Lines Of Code (Source Code)	7947
# Unit Tests	266
Test Coverage	90.28%

In the following sections, we discuss how we tackled the challenges of representing and projecting global types (Section 3.1 and Section 3.2), endpoint implementation (Section 3.3), and refinements (Section 3.4) Lastly, we present some Quality of Life features

that were introduced to assist with developer's workflow (Section 3.5), and we close the chapter with the evaluation (Section 3.6).

3.1 Global Type Representation

To represent global types we considered two approaches:

- Use the Scribble language and extend the Scribble toolchain to target Kotlin (similar to Hu and Yoshida [21]);
- Create a DSL and define the types directly in Kotlin.

The former has the advantage of inheriting Scribble's syntax and validation capabilities but would require developers to learn a new syntax. The latter was the direction chosen for this project: the DSL offers a much more familiar coding environment, and it leverages Kotlin's type system to provide code completion and documentation.

To build multiparty session types, the DSL has the function `globalProtocol(...)` as its entrypoint. The second argument, `protocolBuilder`, is a lambda with receiver (Subsection 2.7.1). This receiver has the operations commonly present in session types:

- `send<T>(from: SKRole, to: SKRole, label: String)`
  Declare that `from` sends a labelled message with type T to `to`;
- `choice(at: SKRole, branches: ChoiceEnv.() -> Unit)`
  Declare that `at` must choose a continuation. The different paths are declared inside the lambda with `branch`:
  - `branch(protocolBuilder: GlobalEnv.() -> Unit).`
- `mu(): RecursionTag`
  Create a recursion point;
- `goto(t: RecursionTag)`
  Declare that the protocol should continue at the point `t` was created.

To better illustrate how sessionkotlin can be used to build session types, we present an implementation of the Two Buyer protocol (Listing 3.1), which was described in the previous section (Figure 2.12). First, we create roles for `A`, `B`, and the `Seller`, which are instances of the provided `SKRole` class (Lines 1-3). Next, we define the protocol using the `globalProtocol` function, passing three arguments: the protocol name (`"TwoBuyer"`), whether the callbacks API should be generated, and the protocol builder, which is a lambda (Subsection 2.7.1). Inside the protocol builder environment, it is possible to invoke the functions detailed above to send messages and offer choices.

Using recursion, we can describe a protocol that sums a variable amount of numbers. In Listing 3.2, `Alice` sends one or more integers to `Bob` before terminating. We use the

Listing 3.1: Two Buyer protocol in sessionkotlin

```
val a = SKRole("ClientA")
val b = SKRole("ClientB")
val seller = SKRole("Seller")

globalProtocol("TwoBuyer", true) {
    send<String>(a, seller, "Id")
    send<Int>(seller, a)
    send<Int>(seller, b)
    send<Int>(a, b)
    choice(b) {
        branch {
            send<String>(b, seller, "Address")
            send<LocalDate>(seller, b, "Date")
            send<LocalDate>(b, a, "Date")
        }
        branch {
            send<Unit>(b, seller, "Quit")
            send<Unit>(b, a, "Quit")
        }
    }
}
```

tag created by the `mu()` call (Line 2) as the argument of the `goto` function (Line 7). The message in Line 10 is needed to propagate the result of the choice to `Bob`: this and other restrictions are detailed in Section 3.2.

Note that these protocol definitions are much more flexible than the standard. In particular, when defining a choice, developers only need to specify the role that makes the choice.

```
globalProtocol("SumProtocol") {
    val t = mu()

    choice(a) {
        branch {
            send<Int>(a, b, "Add")
            goto(t)
        }
        branch {
            send<Unit>(a, b, "Quit")
            send<Int>(b, a) // result
        }
    }
}
```

Listing 3.2: Recursive protocol

```
val quitBranch: GlobalProtocol = {
    send<Unit>(a, b, "Quit")
    send<Int>(b, a) // result
}

globalProtocol("SumProtocol") {
    val t = mu()

    choice(a) {
        branch {
            send<Int>(a, b, "Add")
            goto(t)
        }
        branch {
            quitBranch()
        }
    }
}
```

Listing 3.3: Protocol decomposition

Listing 3.3 showcases the possibility of extracting parts of the protocol. The variable declared in Lines 1-4, when invoked inside the `Quit` branch, has the same effect of the

Lines 10-11 of Listing 3.2. The `GlobalProtocol` type is an alias of a function type that has a global protocol as its receiver, and that is why it is invoked inside the global protocol declaration.

In exchange for the ability to declare global types in the DSL, we lose the ability to validate it statically: the lambda inside `globalProtocol` is not available at compile time. But, in practice, we have two compilation moments: one to compile the global type (and generate local APIs) and one to compile the code that uses the generated APIs. By validating the global type before generating local APIs, we ensure that client code always uses valid types.

3.2 Global Type Projection & Validation

The projection is, in practice, similar to the one shown in Figure 2.7. Due to our global type definitions being much more flexible than the standard definition (Figure 2.6), the implementation is a bit more sophisticated. We perform the following verifications and optimizations:

1. When sending messages:
    a) The sender must be different from the receiver;
    b) If the generation of callbacks API is requested, the sequence {action, label, sender, receiver} must be unique.
    c) The sender must "know" the message labels used in the refinement condition, if present;
    d) All conditions must be parseable.
2. In choices:
    a) A role can be *enabled* or *disabled*.
    b) The choice subject (the role that makes the choice) starts as *enabled*. All the other roles become *disabled*.
    c) Disabled roles become *enabled* upon receiving a message;
    d) A role must be *enabled* in exactly zero or in every branch;
    e) A role must be *enabled* by the same role in every branch;
    f) Branches that describe the same local type as another branch are locally erased.
    g) If all choice branches are empty, the choice is locally erased;
    h) If a *disabled* role sends a message, is the subject of a choice, or if some branch ends with a recursion call, its behavior must be the same for all branches.
3. On recursions:

a) Recursion definitions that are not used are locally erased;

b) If a role does not send, receive, or choose after a recursion definition, the recursion is locally erased;

This transitive notion of activation represents the roles' knowledge of the outcome of choices. The actions of *enabled* and *disabled* roles are the basis for choice validation. To clarify these rules and their purpose consider the following examples, in which we refer to the roles `a`, `b`, and `c` as Alice, Bob, and Charlie, respectively.

Listing 3.4: Unfinished role

```
1  globalProtocol("UnfinishedRoleExample") {
2      choice(a) {
3          branch {
4              send<Int>(a, b, "b1")
5              send<Int>(a, c, "b1")
6          }
7          branch {
8              send<Int>(a, b, "b2")
9              // Charlie hanging
10         }
11     }
12 }
```

```
1  [main] ERROR GlobalEnv - Exception while projecting onto Charlie
2  Exception in thread "main" UnfinishedRolesException: Unfinished roles: Charlie.
```

Listing 3.4 declares a protocol in which Charlie only receives a message in the first branch. As expected, an exception is thrown describing the problem: a role is *not finished*: Charlie cannot tell if it should wait for Alice's message or if the protocol has terminated. Rule 2d, which ensures that there are no *unfinished roles*, is violated.

The protocol declared in Figure 3.5 contains an *inconsistent* choice: In the first branch, Charlie receives it's first message from Bob, while in the second branch it is from Alice. Rule 2e, which guarantees *consistent choices*, is broken and the exception details exactly that: Charlie is activated by two different roles (Bob and Alice).

In Listing 3.6, Charlie does not know the outcome of the choice (he is *disabled*) and thus cannot determine whether to send the message to Bob or not. This example breaks rule 2h, which guarantees that local role behavior does not depend on the outcome of unknown choices. In contrast, the protocol described in Listing 3.7 is perfectly fine: Charlie presents the same behaviour in both branches and the choice is locally collapsed due to rule 2f. Charlie's local type is simply $!\langle Int, Bob\rangle.end$.

As for recursion, consider Listing 3.8. This protocol would appear to break rule 2h but we can combine rules 3b and 2g: the first allows unguarded recursions to be erased from local types, while the second erases empty choices. In this example, Charlie does not need to distinguish between branches because he does not "do anything"after the recursion definition in Line 3. The local type becomes $?\langle Int, Bob\rangle.end$. In contrast, the

Listing 3.5: Inconsistent choice

```
globalProtocol("InconsistentProtocol") {
    choice(a) {
        branch {
            send<Int>(a, b, "b1")
            send<Int>(b, c, "b1") // Charlie activated by Bob
        }
        branch {
            send<Int>(a, b, "b2")
            send<Int>(a, c, "b2") // Charlie activated by Alice
        }
    }
}
```

```
[main] ERROR GlobalEnv - Exception while projecting onto Charlie
Exception in thread "main" InconsistentExternalChoiceException: Inconsistent external
↪  choice: role Charlie activated by [Bob, Alice]
```

Listing 3.6: Role not enabled

```
globalProtocol("Protocol1") {
    choice(a) {
        branch {
            send<Int>(a, b, "b1")
            send<Int>(c, b)  // Charlie not enabled
        }
        branch {
            send<Int>(a, b, "b2")
        }
    }
}
```

```
[main] ERROR GlobalEnv - Exception while projecting onto Charlie
Exception in thread "main" RoleNotEnabledException: Role Charlie not enabled.
```

Listing 3.7: Collapsable choice

```
globalProtocol("Protocol1") {
    choice(a) {
        // Charlie not activated in any branch
        branch {
            send<Int>(a, b, "b1")
            send<Int>(c, b)
        }
        branch {
            send<Int>(a, b, "b2")
            send<Int>(c, b)
        }
    }
}
```

protocol displayed in Listing 3.9 is invalid because it breaks rule 2h. We cannot apply rule 3b because Charlie sends an integer to Alice after the recursion definition (Line 4).

Listing 3.8: Erased recursion

```
globalProtocol("Protocol1") {
    send<Int>(b, c)
    val t = mu()
    choice(a) {
        branch {
            send<Int>(a, b, "b1")
            goto(t) // Charlie not enabled
        }
        branch {
            send<Int>(a, b, "b2")
            send<Int>(b, a)
        }
    }
}
```

Finally, the rule 1b prevents name clashing of generated classes and methods.

Listing 3.9: Role not enabled

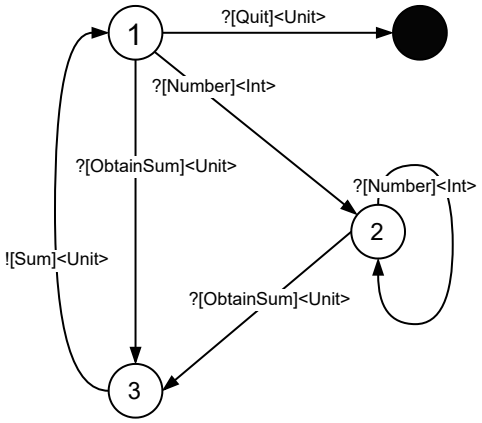
```
globalProtocol("Protocol1") {
    send<Int>(b, c)
    val t = mu()
    send<Int>(c, a) // Charlie sends a message
    choice(a) {
        branch {
            send<Int>(a, b, "b1")
            goto(t) // Charlie not enabled
        }
        branch {
            send<Int>(a, b, "b2")
            send<Int>(b, a)
        }
    }
}
```

```
[main] ERROR GlobalEnv - Exception while projecting onto Charlie
Exception in thread "main" RoleNotEnabledException: Role Charlie not enabled.
```

Once the local types are obtained, similarly to Scribble [59], they are transformed in a Finite State Machine (FSM) to perform simplifications, detect non-deterministic states, and streamline the API generation. Consider the protocol in Figure 3.2 that describes the interactions between two participants, a client and a server. The figure also shows the simplified FSM representation of the server's behaviour, with the receiver and sender omitted. The initial state reflects the ability to receive the *Quit* message from the first branch, and both *Number* and *ObtainSum* from the nested choice.

Figure 3.2: Recursive adder protocol and server FSM representation

```
globalProtocol("RecAdder") {
    val t1 = mu()
    choice(c) {
        branch {
            send<Unit>(c, s, "Quit")
        }
        branch {
            val t2 = mu()
            choice(c) {
                branch {
                    send<Int>(c, s, "Number")
                    goto(t2)
                }
                branch {
                    send<Unit>(c, s, "ObtainSum")
                    send<Int>(s, c, "Sum")
                    goto(t1)
                }
            }
        }
    }
}
```



3.3 Endpoint Implementation

We offer two alternatives for endpoint implementation: a fluent API and a callback-based approach: both can be used interchangeably but provide different guarantees. To generate endpoint code, we use the kotlinpoet library [37], which provides an API for generating Kotlin source files.

3.3.1 Communication

Endpoint communication can be carried out via channels and/or sockets. We provide the following communication API:

- `request(role: SKGenRole, hostname: String, port: Int)`
- `accept(role: SKGenRole, port: Int)`
- `accept(role: SKGenRole, serverSocket: SKServerSocket)`
- `bind(port: Int): SKServerSocket`
- `connect(role: SKGenRole, chan: SKChannel)`
- `wrap(role: SKGenRole, wrapper: SocketWrapper)`

Where `SKGenRole` is a superclass that all generated roles inherit from, and `SKChannel` is a class that wraps two Kotlin channels to allow bidirectional communication. A single `SKChannel` instance is shared between two endpoints. The `accept(..., port: Int)` operation will implicitly create a socket and bind it. For the socket to persist across `SKMPEndpoint` instances, it is possible to obtain an `SKServerSocket` by explicitly calling the static `bind`

method. The `wrap` method is used to wrap an existing socket connection: we offer an implementation of TLS, `TLSSocketWrapper`.

We use the suspending-style socket API offered by the Ktor framework [39], which internally uses java.nio. This allows us to create a unified communication API that supports both sockets and channels.

3.3.2 Fluent API

Listing 3.10: Seller endpoint implementation (fluent API)

```
runBlocking {
    val chanClientASeller = SKChannel()
    val chanClientBSeller = SKChannel()
    launch {
        SKMPEndpoint().use { e ->
            e.connect(ClientA, chanClientASeller)
            e.connect(ClientB, chanClientBSeller)

            val productId = ""
            val address = ""

            TwoBuyerSeller1(e)
                .receiveFromClientA { productId = it }
                .sendToClientA(getPriceByName(productId))
                .sendToClientB(getPriceByName(productId))
                .branch()
                .let {
                    when (it) {
                        is TwoBuyerSeller4_AddressInterface -> it
                            .receiveFromClientB { address = it }
                            .sendToClientB(calculateDelivery(address))
                        is TwoBuyerSeller4_QuitInterface -> it
                            .receiveFromClientB()
                    }
                }
        }
    }
}
```

To support this style, a class is created for each state. The class corresponding to the initial state is public, acting as the entry point. Each method returns an instance to the next state, making it possible to chain the calls. These methods are guarded: should they be called more than once, an exception is thrown and no I/O is performed, preserving linearity.

Figure 3.10 shows an implementation of Seller endpoint of the two buyer protocol using this style. It starts by creating an `SKMPEndpoint` and connecting to both clients via channels. It then creates an instance of the initial state (`TwoBuyerSeller1`), passing the endpoint as an argument, and executes the protocol by calling the methods. Note that the I/O methods can suspend, so they can only be called inside coroutines.

3.3.3 Callbacks

The idea behind this approach is to invert the control and have the library call user-defined code. Similarly to Zhou et al. [60], the developer defines callbacks and the backend invokes them as the protocol progresses. The main advantage is that, as the user cannot explicitly send or receive messages, linearity is achieved *by construction*, and we avoid the runtime verifications that the fluent approach need. To generate an API in this style, simply pass `True` as the `callbacks` argument of the `globalProtocol` function.

Listing 3.11 shows how we can implement the seller endpoint of the Two Buyer protocol using the callbacks API. The library generates an interface with the callback signatures (`TwoBuyerCallbacksSeller`) and an endpoint (`TwoBuyerCallbackEndpointSeller`). The developer implements the callbacks and pass them to the endpoint as an argument. The endpoint provides methods to connect to other endpoints, just like `SKMPEndpoint`, and a method that starts the execution of the protocol, `start`.

Listing 3.11: Seller endpoint implementation (callbacks API)

```
runBlocking {
    val chanClientASeller = SKChannel()
    val chanClientBSeller = SKChannel()
    launch {
        val productId = ""
        val address = ""

        val cs = object : TwoBuyerSellerCallbacks {
            override fun receiveIdFromClientA(v: String) { productId = v }
            override fun sendToClientA(): Int = getPriceByName(productId)
            override fun sendToClientB(): Int = getPriceByName(productId)
            override fun receiveAddressFromClientB(v: String) { address = v }
            override fun receiveQuitFromClientB() {}
            override fun sendDateToClientB(): LocalDate = calculateDelivery(address)
        }
        TwoBuyerSellerCallbacksEndpoint(cs).use {
            it.connect(ClientA, chanClientASeller)
            it.connect(ClientB, chanClientBSeller)
            it.start()
        }
    }
}
```

3.3.4 Safety Guarantees

The callbacks API guarantees linearity by construction as a result of I/O not being called directly by the developer. Endpoint code that uses the fluent API contains runtime checks. In any case, endpoint code is only generated after global type validation: the APIs statically define the allowed I/O and require exhaustive handling of all branches, resulting in protocol fidelity and communication safety.

3.4 Refinements

The DSL also supports type refinements: it is possible to define logical expressions to constrain the exchanged data. We show, in Figure 3.12, how we could extend the global type of the Two Buyer protocol to include these refinements. The first condition (Line 8) ensures that both clients receive the same value for the price of the product; the second condition (Line 9) guarantees that Client A does not pay the total amount.

Listing 3.12: Refined Two Buyer protocol in sessionkotlin

```
1  val a = SKRole("ClientA")
2  val b = SKRole("ClientB")
3  val seller = SKRole("Seller")
4
5  globalProtocol("TwoBuyer") {
6    send<String>(a, seller, "Id")
7    send<Int>(seller, a, "valA")
8    send<Int>(seller, b, "valB", "valA == valB")
9    send<Int>(a, b, "proposal", "proposal <= valA")
10   choice(b) {
11     branch {
12       send<String>(b, seller, "Address")
13       send<LocalDate>(seller, b, "Date")
14       send<LocalDate>(b, a, "Date")
15     }
16     branch {
17       send<Unit>(b, seller, "Quit")
18       send<Unit>(b, a, "Quit")
19     }
20   }
21 }
```

The expressions can contain integers, floating-point numbers, string literals, and message labels. It also supports basic arithmetic operations (sum, subtraction), comparisons, and boolean operators (and, or, implication). The parser was built with the parser combinator library better-parse [5]. For a full grammar specification check out Figure 3.3.

For an expression to be valid, all variables must be visible *locally*. In Listing 3.13, the condition is not valid because `val1` does not exist in the branch.

In Listing 3.14 we have two conditions. The first, `val2 == val1`, is valid because Bob has knowledge of both messages and only the message sender enforce the refinement. In opposition, `val3 == val1` is not valid since Charlie does not know the value of `val1`.

It is also needed to check for satisfiability. This can be done by employing an external solver (for example, an SMT solver) [45, 60, 13]. We use the Z3 theorem prover [54], as it supports integers, floating-point numbers and strings, through the API provided by the java-smt library [3].

As an example, Listing 3.15 defines an expression that is not satisfiable: `val2` cannot be greater than zero and at the same time have the same value of `val1`. This information is reported by the library through an exception with information about the offending condition.

Figure 3.3: Refinement expressions grammar

$$
\begin{aligned}
ImplChain &\triangleq OrChain -> ImplChain \\
&\quad | \; OrChain \\
\\
OrChain &\triangleq OrChain \;\&\&\; AndChain \\
&\quad | \; AndChain \\
\\
AndChain &\triangleq AndChain \,||\, Bool \\
&\quad | \; Bool \\
\\
Term &\triangleq \\
&\quad | \; -Term \\
&\quad | \; (Expr) \\
&\quad | \; \text{'string' // String literal} \\
&\quad | \; \text{string // Variable} \\
&\quad | \; \text{float} \\
&\quad | \; \text{integer}
\end{aligned}
\qquad
\begin{aligned}
Bool &\triangleq \\
&\quad | \; \text{true} \mid \text{false} \\
&\quad | \; !\, Bool \\
&\quad | \; Expr == Expr \\
&\quad | \; Expr\,! = Expr \\
&\quad | \; Expr < Expr \\
&\quad | \; Expr <= Expr \\
&\quad | \; Expr > Expr \\
&\quad | \; Expr >= Expr \\
&\quad | \; (ImplChain) \\
\\
Expr &\triangleq \\
&\quad | \; Expr + Term \\
&\quad | \; Expr - Term \\
&\quad | \; Term
\end{aligned}
$$

Listing 3.13: Unknown variable (scope)

```
globalProtocol("RefinedProtocol1") {
  choice(a) {
    branch {
      send<Int>(a, b, "val1")
    }
    branch {
      send<Int>(a, b, "val2", "val2 != val1")
    }
  }
}
```

```
[main] ERROR GlobalEnv - Exception while projecting onto Alice
Exception in thread "main" UnknownMessageLabelException: Role Alice cannot see some
↪  message labels: [val1]
```

Any problems with name visibility or satisfiability cause the global type to be invalid and prevent local types from being generated. All refinement conditions materialize to assertions that are inserted within the endpoint code.

Listing 3.14: Unknown variable (roles)

```
globalProtocol("RefinedProtocol2") {
  send<Int>(a, b, "val1")
  send<Int>(b, c, "val2", "val2 == val1")
  send<Int>(c, a, "val3", "val3 == val1")
}
```

```
[main] ERROR GlobalEnv - Exception while projecting onto Charlie
Exception in thread "main" UnknownMessageLabelException: Role Charlie cannot see some
↪  message labels: [val1]
```

Listing 3.15: Unsatisfiable refinement

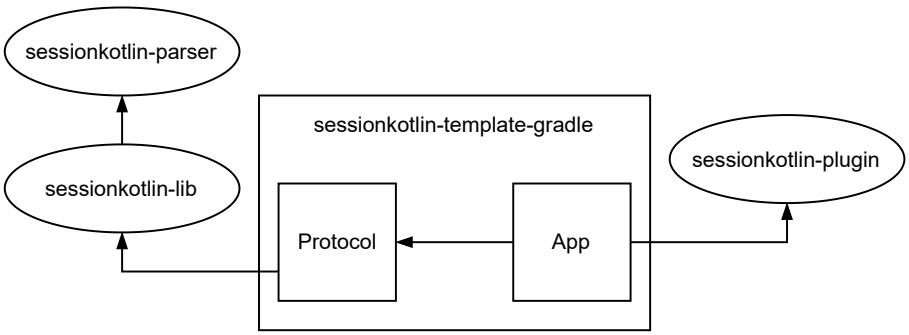
```
globalProtocol("RefinedProtocol3") {
  send<Int>(a, b, "val1", "val1 < 0")
  send<Int>(a, b, "val2", "val2 > 0 && val2 == val1")
}
```

```
Exception in thread "main" UnsatisfiableRefinementsException: (and (< val1 0) (> val2 0)
↪  (= val2 val1))
```

3.5 Quality of Life Features

The library is available as a Maven artifact and can be added as a dependency to Gradle or Maven projects. We created two project templates on GitHub: one that uses Gradle [1] and one with Maven [2]. The former is recommended for a couple of reasons. It is a multi-project build [3]: one project contains the protocol definition, and the other the main app. With this structure, we split the two compilation steps discussed in Section 3.1, and it becomes possible to change the protocol definition as needed without commenting out the main application code. Additionally, it uses a plugin we built that simplifies the build configuration (the solver library requires some files to be in a special folder).

Figure 3.4: Template Gradle project



[1] https://github.com/d-costa/sessionkotlin-template-gradle/tree/thesis
[2] https://github.com/d-costa/sessionkotlin-template-maven/tree/thesis
[3] https://docs.gradle.org/current/userguide/multi_project_builds.html

The Gradle project is detailed in Figure 3.4. The *Protocol* subproject declares a dependency on the sessionkotlin-lib artifact, which itself depends on the sessionkotlin-parser module. The *App* subproject has access to the library and generated code via its dependency on *Protocol*. Finally, the plugin takes care of all the necessary configurations.

```
fun aux(sender: SKRole, receiver: SKRole): GlobalProtocol = {
    send<Int>(sender, receiver)
}
globalProtocol("Protocol1") {
    aux(a, b)() // send<Int>(a, b)
    aux(b, a)() // send<Int>(b, a)
}
```

Listing 3.16: Parameterized protocol decomposition

Protocol decomposition was also an important feature to have in the DSL. In addition to what was shown in Listing 3.3, it is possible to go one step further and create parameterized protocols. For example, in Listing 3.16, we define a higher-order function `aux` that takes two roles as arguments and returns a function that returns a protocol (`GlobalProtocol` is a type alias of a function type that has a global protocol as its receiver). This allows developers to extract and reuse sections of a big protocol.

3.6 Evaluation

In this section we discuss potential limitations present in the library, show how SMTP can be implemented using our tools, and examine the results of micro-benchmarks.

3.6.1 Limitations

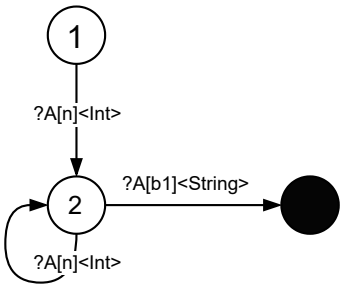
Some protocol definitions that could represent valid global types are rejected by sessionkotlin. Consider the protocol defined in Figure 3.5, between Alice (`a`), Bob (`b`), and Charlie (`c`). Charlie starts the protocol by receiving a message `n` from Alice. Then, depending on Alice's choice, either receives a message labeled `b1` or recurs. Unfortunately, rule 2h causes this interaction to be invalid: Charlie is not aware of Alice's choice at the point of recursion. However, if we *unroll* the recursion, it is obvious that Charlie can infer Alice's choice: the `n` message, if received, means that Alice chose the second branch. Scribble [59] implements this *unrolling* and correctly validates this protocol. Figure 3.5 illustrates the possible states that Charlie could go through.

On the other hand, protocols such as the one declared in Figure 3.7 are invalid through Scribble's rules: only enabled roles are allowed to send messages. When projecting Charlie's local type, our library erases the external choice since all branches describe the same behaviour, resulting in a valid global protocol.

As previously stated, protocol validation is performed during runtime (Section 3.2). We argue that relying on runtime checks is reasonable for two reasons. First, malformed protocols are never allowed since the library will never generate APIs from them. Second,

Figure 3.5: Rejected protocol and possible Charlie's FSM representation

```
1  globalProtocol("Protocol") {
2      val t = mu()
3      send<Int>(a, c, "n")
4      choice(a) {
5          branch {
6              send<String>(a, b, "b1")
7              send<String>(a, c, "b1")
8          }
9          branch {
10             send<Int>(a, b, "b2")
11             goto(t)
12         }
13     }
14 }
```



the usage discipline complements the runtime verifications: the fluent API guides the developer through session states with the return types, and the autocomplete feature present in every modern IDE avoids the need to know about function name generation and the possible outcomes of a choice. Linearity is enforced through runtime assertions, and any attempt to break it results in an exception. The callbacks API provides linearity by construction but requires developers to define the callbacks while keeping in mind the message labels defined earlier. This strategy allows global protocol definition through type-safe builders, which provide an intuitive and idiomatic way to create multiparty session types while simultaneously taking advantage of the host language ecosystem.

The refinements have some restrictions as well: payload types are limited to `String`, `Long` and `Double`, while `Int`, `Short`, and `Byte` payloads are promoted to `Long`. Additionally, the expressions are dynamically typed, which means that type compatibility is checked during runtime, while the protocol is being validated. Therefore, invalid expressions such as `"'a' > 0"` (string literal `a` must be greater than zero) are not statically flagged as such and result in a runtime exception.

3.6.2 Case Study - SMTP

We implemented a simplified version of SMTP [31], the internet standard for mail transport, together with a client[4] that is capable of communicating with external SMTP servers such as Gmail's SMTP server.

An execution log is show in Listing 3.17. The session is initiated when the client opens a connection to the server and receives its welcome message (Line 1). In this example, we connect to Gmail's SMTP server. The client sends the ehlo message (Line 2), to which the server replies with the supported extensions (Lines 3-10). The client continues by signaling the intention to use the TLS extension [18], and the server acknowledges (Lines 11-12). At this point, the TLS handshake is carried out: the unsecured connection is wrapped within a `TLSSocketWrapper` object that handles the handshake and transparently encrypts

[4] https://github.com/d-costa/sessionkotlin/tree/thesis/evaluation/smtp

Listing 3.17: SMTP log

```
1  SKMPEndpoint - Received: 220 smtp.gmail.com ESMTP c13-20020a056000104d00b0021cf31e1f7csm1091022wrx.102 - gsmtp
2  SKMPEndpoint - Sent    : Ehlo com.github.d_costa
3  SKMPEndpoint - Received: 250-smtp.gmail.com at your service, [IP]
4  SKMPEndpoint - Received: 250-SIZE 35882577
5  SKMPEndpoint - Received: 250-8BITMIME
6  SKMPEndpoint - Received: 250-STARTTLS
7  SKMPEndpoint - Received: 250-ENHANCEDSTATUSCODES
8  SKMPEndpoint - Received: 250-PIPELINING
9  SKMPEndpoint - Received: 250-CHUNKING
10 SKMPEndpoint - Received: 250 SMTPUTF8
11 SKMPEndpoint - Sent    : StartTLS
12 SKMPEndpoint - Received: 220 2.0.0 Ready to start TLS
13 TLSSocketWrapper - Client: TLS Handshake complete
14 TLSSocketWrapper - Client: Protocol: TLSv1.3
15 TLSSocketWrapper - Client: CipherSuite: TLS_AES_256_GCM_SHA384
16 SKMPEndpoint - Sent    : Ehlo com.github.d_costa
17 SKMPEndpoint - Received: 250-smtp.gmail.com at your service, [IP]
18 SKMPEndpoint - Received: 250-SIZE 35882577
19 SKMPEndpoint - Received: 250-8BITMIME
20 SKMPEndpoint - Received: 250-AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH
21 SKMPEndpoint - Received: 250-ENHANCEDSTATUSCODES
22 SKMPEndpoint - Received: 250-PIPELINING
23 SKMPEndpoint - Received: 250-CHUNKING
24 SKMPEndpoint - Received: 250 SMTPUTF8
25 SKMPEndpoint - Sent    : Auth LOGIN
26 SKMPEndpoint - Received: 334 Username:
27 SKMPEndpoint - Sent    : ZGNvc3RhLnNtdHBAZ21haWwuY29t
28 SKMPEndpoint - Received: 334 Password:
29 SKMPEndpoint - Sent    : [password]
30 SKMPEndpoint - Received: 235 2.7.0 Accepted
31 SKMPEndpoint - Sent    : Mail from:<dcosta.smtp@gmail.com>
32 SKMPEndpoint - Received: 250 2.1.0 OK c13-20020a056000104d00b0021cf31e1f7csm1091022wrx.102 - gsmtp
33 SKMPEndpoint - Sent    : Rcpt to:<dcosta.smtp@gmail.com>
34 SKMPEndpoint - Received: 250 2.1.5 OK c13-20020a056000104d00b0021cf31e1f7csm1091022wrx.102 - gsmtp
35 SKMPEndpoint - Sent    : Data
36 SKMPEndpoint - Received: 354  Go ahead c13-20020a056000104d00b0021cf31e1f7csm1091022wrx.102 - gsmtp
37 SKMPEndpoint - Sent    : message-id:<1657795062413AE98FCE0F8306893CBBABDDCC5B567F7@com.github.d_costa>
38 SKMPEndpoint - Sent    : from:<dcosta.smtp@gmail.com>
39 SKMPEndpoint - Sent    : to:<dcosta.smtp@gmail.com>
40 SKMPEndpoint - Sent    : subject:Hello!
41 SKMPEndpoint - Sent    : Hello world,
42 SKMPEndpoint - Sent    :
43 SKMPEndpoint - Sent    : Lorem ipsum dolor sit amet, consectetur adipiscing elit.
44 SKMPEndpoint - Sent    : Cras hendrerit posuere augue, ut pulvinar nulla semper ut.
45 SKMPEndpoint - Sent    : .
46 SKMPEndpoint - Received: 250 2.0.0 OK  1657795063 c13-20020a056000104d00b0021cf31e1f7csm1091022wrx.102 - gsmtp
47 TLSSocketWrapper - Client: Closing connection...
48 TLSSocketWrapper - Client: Connection closed.
```

and decrypts messages. The ehlo message is repeated, but this time the server sends a list of authentication methods [52] (Line 20). The client proceeds by authenticating itself to the server using the Login method (Lines 25-30). Next, the client signals the beginning of a mail transaction by sending a Mail message (Line 31). After setting the necessary headers and transmitting the mail body, the transaction finishes when the termination signal is emitted (the full stop, on Line 45). Finally, the server indicates that the message was accepted and the connection is closed.

In addition to being a valuable exercise in expressiveness, the development of the SMTP client led to several changes and improvements to the library. The most notable one is that initial versions of the DSL had labeled branches instead of labeled messages and that API generation was based on the local types. The label corresponding to the selected branch was sent on the first message of that branch, which caused a type of construction to be problematic. Consider the excerpt of an initial SMTP shown in Figure 3.18, which illustrates the server's possible answers to client authentication. If accepted, the server

Listing 3.18: Excerpt of an early SMTP implementation

```
choice(server) {
    branch("Success") {
        // Authentication succeeded
        send<C235>(server, client)
        mail() // The mail transaction
    }
    branch("AuthFailed") {
        // Authentication unsuccessful
        val t = mu()
        choice(server) {
            branch("C535") {
                send<C535>(server, client)
            }
            branch("C535H") {
                send<C535H>(server, client)
                goto(t)
            }
        }
    } // Other branches omitted
}
```

responds with a `C235` message. Otherwise, it can send zero or more `C535H` messages before finalizing with `C535`. Note that it is not possible to flatten the choices due to the presence of recursion in the inner choice. The issue arises when deciding what label is sent alongside the messages inside the `AuthFailed` branch: should we discard the outer label and use only `C535H` and `C535`, or combine them (`AuthFailed_C535`)? If all endpoints were using the APIs generated by the library it would pose no problem, but when dealing with an SMTP server that does not know about any labels, we concluded that the most intuitive design would be to only label the messages.

Additionally, instead of generating the APIs directly from the local types, we added finite state machines as an intermediate representation: transforming the inherently recursive local types into sets of states and transitions allows us to analyse and simplify the state graph before API generation. The finite state machine, unlike the local type, maps directly to the APIs: in the case of the fluent variant, states become classes and the transitions specify their methods; for the callbacks API, the states map to functions.

Finally, the need to secure an ongoing TCP connection prompted the development of `SKMPEndpoint`'s `wrap` functionality: it creates an interface between the application and transport layers that modifies inbound and outbound data as needed. The library implements a TLS wrapper, but any class that implements the `SocketWrapper` API can be employed.

A simplified definition of the protocol is shown in Listing 3.19. A class was created for each kind of message (`Ehlo`, `C220`, etc.), and the message codes, used as labels, are statically defined within the `Code` class.

```
globalProtocol("SMTP") {
    choice(server) {
        branch {
            // Service ready
```

```
5               send<C220>(server, client, Code.C220)
6               ehlo(tls)()
7           }
8           branch {
9               // Transaction failed
10              send<C554>(server, client, Code.C554)
11          }
12      }
13  }
14
15  fun ehlo(continuation: GlobalProtocol): GlobalProtocol = {
16      choice(client) {
17          branch {
18              send<Ehlo>(client, server, Code.Ehlo)
19              val t = mu()
20              choice(server) {
21                  branch {
22                      send<C250Hyphen>(server, client, Code.C250Hyphen)
23                      goto(t)
24                  }
25                  branch {
26                      send<C250>(server, client, Code.C250)
27                      continuation()
28                  }
29              }
30          }
31          branch {
32              clientQuit()
33          }
34      }
35  }
36
37  val tls: GlobalProtocol = {
38      choice(client) {
39          branch {
40              send<StartTLS>(client, server, Code.TLS)
41              send<C220>(server, client, Code.C220)
42              // Do TLS handshake here, e.g.:
43              // endpoint.wrap(Server, TLSSocketWrapper(ConnectionEnd.Client))
44              ehlo(auth)()
45          }
46          branch {
47              clientQuit()
48          }
49      }
50  }
51
52  val auth: GlobalProtocol = {
53      choice(client) {
54          branch {
55              send<AuthLogin>(client, server, Code.Auth)
56              send<C334>(server, client)
57              send<AuthUsername>(client, server)
58              send<C334>(server, client)
59              send<AuthPassword>(client, server)
60              choice(server) {
61                  branch {
62                      // Authentication Succeeded
63                      send<C235>(server, client, Code.C235)
64                      mail()
65                  } // 501, 504, 534, 535, and 538 branches ommited
66              }
67          }
68      }
69  }
70
71  val mail: GlobalProtocol = {
72      send<Mail>(client, server, Code.Mail)
73      choice(server) {
```

```
74          branch {
75              // OK
76              send<C250>(server, client, Code.C250)
77              recipients()
78          } // 553 and 530 branches ommited
79      }
80  }
81
82  val recipients: GlobalProtocol = {
83      val t = mu()
84      choice(client) {
85          branch {
86              send<RCPT>(client, server, Code.RCPT)
87              choice(server) {
88                  branch {
89                      // OK
90                      send<C250>(server, client, Code.C250)
91                      goto(t)
92                  }
93                  branch {
94                      branch550()
95                  }
96              }
97          }
98          branch {
99              data()
100         }
101     }
102 }
103
104 val branch550: GlobalProtocol = {
105     // Requested action not taken: mailbox unavailable (e.g., mailbox
106     // not found, no access, or command rejected for policy reasons)
107     val t = mu()
108     choice(server) {
109         branch {
110             send<C550>(server, client, Code.C550)
111         }
112         branch {
113             send<C550Hyphen>(server, client, Code.C550Hyphen)
114             goto(t)
115         }
116     }
117 }
118 val data: GlobalProtocol = {
119     send<Data>(client, server, Code.Data)
120     send<C354>(server, client, Code.C354)
121     bodyHeaders()
122
123     val t = mu()
124     choice(client) {
125         branch {
126             // Add a line
127             send<DataLine>(client, server, Code.DataLine)
128             goto(t)
129         }
130         branch {
131             // End data
132             send<DataOver>(client, server, Code.DataOver)
133             choice(server) {
134                 branch {
135                     // Ok
136                     send<C250>(server, client, Code.C250)
137                 }
138                 branch {
139                     branch550()
140                 }
141             }
142         }
```

```
143     }
144 }
145
146 val bodyHeaders: GlobalProtocol = {
147     send<MessageIdHeader>(client, server)
148     send<FromHeader>(client, server)
149     send<ToHeader>(client, server)
150     send<SubjectHeader>(client, server)
151 }
152
153 val clientQuit: GlobalProtocol = {
154     send<Quit>(client, server, Code.Quit)
155     send<C221>(server, client, Code.C221)
156 }
```

Listing 3.19: SMTP definition in sessionkotlin

3.6.3 Benchmarks

We created a set of micro-benchmarks that measure the throughput of the two implemented APIs, fluent and callbacks, and the two backends, sockets and channels. Additionally, we included a third variant that implements the protocol by hand, *i.e.* without sessionkotlin, to evaluate the library's overhead.

Testing was done with the Java Microbenchmark Harness [28] through the JMH Gradle plugin [30]. Instructions on how to reproduce the presented results are available on the project's repository [5].

We implemented three protocols that, together, use all of the features offered by the DSL: choice, recursion, and refinements. Figure 3.6 illustrates two of them: Adder describes the behaviour of a recursive server that receives two integers and returns their sum, while TwoBuyer is a recursive variant of our running example. The third protocol, AdderRefined, is not present in the figure. It is identical to Adder with the addition of a refinement in Line 10 that asserts that the sum is correct: `"Sum == V1 + V2"`. The implemented endpoints of all protocols loop 1000 times over `t` before quitting.

A comparison with Scribble's Java runtime was considered, but initial testing yielded strange results: throughput was reported to be an order of magnitude worse than sessionkotlin. Through personal communication with the library authors, we learned that the Java runtime is still in an experimental state and is event-driven, focused on scalability. For these reasons, the comparison with Scribble's runtime was dropped.

The results were obtained by averaging the throughput over three trials, each with five warmup iterations and five measurement iterations. The warmup iterations ensure that the JVM reaches a steady state before any measurements. Each trial is executed in a freshly forked JVM, to avoid reusing profile-guided optimizations from previously run benchmarks. The error corresponds to half of a 99.9% confidence interval.

We tasked sessionkotlin to generate fluent and callback-based endpoint APIs for each protocol shown in Figure 3.6. For each API, we implemented a version that uses sockets and one that uses channels. We also added a third variant, *Handwritten*, that implements

[5] https://github.com/d-costa/sessionkotlin/tree/thesis/evaluation/benchmark

Figure 3.6: Protocols used for benchmarking

```
globalProtocol("Adder", true) {
    val t = mu()
    choice(client) {
        branch {
            send<Unit>(client, server, "Quit")
        }
        branch {
            send<Int>(client, server, "V1")
            send<Int>(client, server, "V2")
            send<Int>(server, client, "Sum")
            goto(t)
        }
    }
}
```

```
globalProtocol("TwoBuyer", true) {
    val t = mu()
    choice(a) {
        branch {
            send<String>(a, seller, "Id")
            send<Int>(seller, a, "Price")
            send<Int>(seller, b, "Price")
            send<Int>(a, b, "aShare")
            choice(b) {
                branch {
                    send<String>(b, seller, "Address")
                    send<Date>(seller, b, "Date")
                    send<Date>(b, a, "Date")
                    goto(t)
                }
                branch {  // Unused branch
                    send<Unit>(b, seller, "Reject")
                    send<Unit>(b, a, "Reject")
                    goto(t)
                }
            }
        }
        branch {
            send<Unit>(a, seller, "Quit")
            send<Unit>(seller, b, "Quit")
        }
    }
}
```

the protocol by hand: it does not use sessionkotlin but uses the same communication primitives (Kotlin channels and the async socket API offered by Ktor[39]). Table 3.2 reports the test results for sockets, while Table 3.3 shows the results for the channels.

Table 3.2: Throughput using sockets (ops/s, higher is better)

Protocol	API	Score	Error
Adder	Callbacks	10.015	± 0.237
Adder	Fluent	9.989	± 0.206
Adder	*Handwritten*	16.139	± 0.552
AdderRefined	Callbacks	9.861	± 0.197
AdderRefined	Fluent	9.886	± 0.212
TwoBuyer	Callbacks	4.128	± 0.052
TwoBuyer	Fluent	4.120	± 0.043
TwoBuyer	*Handwritten*	6.206	± 0.314

Table 3.3: Throughput using channels (ops/s, higher is better)

Protocol	API	Score	Error
Adder	Callbacks	900.015	± 12.899
Adder	Fluent	1000.775	± 18.362
Adder	*Handwritten*	1823.978	± 49.704
AdderRefined	Callbacks	701.728	± 13.518
AdderRefined	Fluent	819.187	± 37.289
TwoBuyer	Callbacks	313.886	± 8.674
TwoBuyer	Fluent	320.949	± 10.549
TwoBuyer	*Handwritten*	759.749	± 3.834

As expected, throughput is inversely proportional to the amount of messages exchanged: TwoBuyer needs roughly double the messages relatively to the Adder protocol. It is also quite clear that channels perform much better than sockets: the throughput of the trials that use channels are orders of magnitude better than the alternative. This is not caused by our library: even the handwritten endpoints follow this trend. Since the channels we use are backed by a linked-list buffer, endpoint code that uses them is, in theory, simpler to optimize than code that uses sockets (either by the Kotlin compiler or during runtime by the JIT compiler). It is also important to note that all socket connections are local: remote connections would be even slower.

Relatively to the refined protocol, AdderRefined, the trials that used the socket backend had a very similar performance to the unrefined variant. On the other hand, the ones that used channels performed about 20% worse. It is possible that, when inlining the assertions, some type of compiler or JIT optimization is lost.

As for the APIs, both fluent and callback-based exhibit similar performance: the linear by construction guarantees that the callback API offers does not introduce any additional overhead. Unfortunately, the two perform worse than their handwritten counterparts. In addition to the runtime linear checks required by the fluent API, endpoint code contains more abstraction layers and serialization. Note that, while the loss in throughput appears to be quite large, the corresponding elapsed time difference is in the order of a hundredth of a second. Nonetheless, we consider that the safety benefits offered by the library outweigh the performance loss.

4

Conclusion

In this thesis we develop the first embedding of session types in Kotlin, and a DSL for multiparty session type definition (Section 3.1). Since one of the main implementation requirements was to provide a reasonable user experience and an idiomatic DSL, we allow a less strict syntax compared to standard MPST theory. We adopt a hybrid verification strategy that combines static and dynamic verifications: while sacrificing fully static guarantees in favor of usability, our approach still protects the user from communication errors and protocol violations. Our generated endpoint APIs (fluent and callback-based) materialize message types as Kotlin types, ensuring communication safety *for free*, provided that all participants use APIs originated from the same protocol. Protocol fidelity is guaranteed by design, in the case of the callbacks API, since message IO is not called directly by the user. In contrast, the fluent API relies on runtime verifications to ensure linearity, throwing an exception upon API misuse (*e.g.* attempting to reuse an endpoint). IDE features such as code completion and static code analysis greatly facilitate the usage of both APIs, freeing the user from being familiar with code generation rules, such as method and class naming, when writing endpoint code.

We provide an implementation that supports both concurrent and distributed programs, allowing connections through TCP sockets and Kotlin channels (Subsection 3.3.1), and offer type refinements (Section 3.4). Our generated endpoint API includes two code styles: fluent (Subsection 3.3.2) and callback-based (Subsection 3.3.3). The first is based on method-chaining and relies on runtime linear checks, while the latter offers linearity by design. Additionally, the provided Gradle plugin [51] greatly simplifies the build configuration by managing solver dependencies, and the project templates offer a quick way to get a project started (Section 3.5).

We demonstrate our library's expressiveness by implementing SMTP and prove protocol compliance by successfully communicating with the Gmail servers (Subsection 3.6.2). Additionally, we show that the generated code has a reasonable performance (Subsection 3.6.3).

As the selected communication backends are unknown at protocol validation, session delegation is not supported. Only channel connections could be delegated, since sending

socket references is impossible.

Listing 4.1: Hypothetical static endpoint declaration

```
@SKMP("Adder.scr", "Server")
class AdderServer

@SKMP("Adder.scr", "Client")
class AdderClient
```

The main topics of future work focus on approximating the performance to the hand-written endpoints, and improving refinement validation, which could include expression simplification (*e.g.* `"a > 2 && true"`) and new supported types (*e.g.* `Boolean` messages).

It would also be interesting to explore an alternative way to define global protocols. In Listing 4.1 we define two (dual) classes, `AdderServer` and `AdderClient`, and annotate them with `@SKMP`. Using an annotation processor such as the Kotlin Symbol Processing API [36], we can generate new classes and extension functions to create an endpoint API (Listing 4.2). The global protocol would be defined in an external file (in this case, `Adder.scr`), and would be processed before compilation takes place.

Listing 4.2: Hypothetical generated code

```
class AdderServer2
class AdderServer3

fun AdderServer.receiveFromClient(buf: SKBuffer<Int>): AdderServer2 {
    // ...
}
fun AdderServer2.receiveFromClient(buf: SKBuffer<Int>): AdderServer3 {
    // ...
}
fun AdderServer3.sendToClient(v: Int): AdderProtocolEnd {
    // ...
}
```

Bibliography

[1] *νScr*. URL: https://github.com/nuscr/nuscr/ (15/6/2022) (cit. on p. 17).

[2] B. Almeida, A. Mordido, and V. T. Vasconcelos. "FreeST: Context-free Session Types in a Functional Language". In: *Electronic Proceedings in Theoretical Computer Science* 291 (Apr. 2019), pp. 12–23. ISSN: 2075-2180. DOI: 10.4204/eptcs.291.2. URL: http://dx.doi.org/10.4204/EPTCS.291.2 (cit. on p. 13).

[3] D. Baier, D. Beyer, and K. Friedberger. "JavaSMT 3: Interacting with SMT Solvers in Java". In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 195–208. ISBN: 978-3-030-81687-2. DOI: 10.1007/978-3-030-81688-9_9. URL: https://doi.org/10.1007/978-3-030-81688-9_9 (cit. on p. 33).

[4] J.-P. Bernardy et al. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093 (cit. on p. 17).

[5] *better-parse*. URL: https://github.com/h0tk3y/better-parse (3/6/2022) (cit. on p. 33).

[6] L. Bocchi et al. "A Theory of Design-by-Contract for Distributed Multiparty Interactions". In: *CONCUR 2010 - Concurrency Theory*. Ed. by P. Gastin and F. Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 162–176. ISBN: 978-3-642-15375-4 (cit. on p. 11).

[7] L. Caires and H. T. Vieira. "Conversation Types". In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by G. Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 285–300. DOI: 10.1007/978-3-642-00590-9_21. URL: https://doi.org/10.1007/978-3-642-00590-9%5C_21 (cit. on p. 5).

[8] M. Carbone, N. Yoshida, and K. Honda. "Asynchronous Session Types: Exceptions and Multiparty Interactions". In: *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. Vol. 5569. LNCS. Springer, 2009, pp. 187–212. DOI: 10.1007/978-3-642-01918-0_5 (cit. on p. 9).

[9] R. F. Chen, S. Balzer, and B. Toninho. *Ferrite: A Judgmental Embedding of Session Types in Rust*. 2022. DOI: 10.48550/ARXIV.2205.06921. URL: https://arxiv.org/abs/2205.06921 (cit. on pp. 14, 17).

[10] M. Coppo et al. "A Gentle Introduction to Multiparty Asynchronous Session Types". In: *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Ed. by M. Bernardo and E. B. Johnsen. Vol. 9104. Lecture Notes in Computer Science. Springer, 2015, pp. 146–178. DOI: 10.1007/978-3-319-18941-3_4. URL: https://doi.org/10.1007/978-3-319-18941-3_4 (cit. on pp. 3, 7).

[11] Z. Cutner and N. Yoshida. "Safe Session-Based Asynchronous Coordination in Rust". In: *Coordination Models and Languages*. Ed. by F. Damiani and O. Dardha. Cham: Springer International Publishing, 2021, pp. 80–89. ISBN: 978-3-030-78142-2 (cit. on p. 17).

[12] Z. Cutner, N. Yoshida, and M. Vassor. *Deadlock-free asynchronous message reordering in Rust with multiparty session types*. 2021. DOI: 10.48550/ARXIV.2112.12693. URL: https://arxiv.org/abs/2112.12693 (cit. on p. 17).

[13] A. Das and F. Pfenning. *Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description)*. Ed. by Z. M. Ariola. 2020. DOI: 10.4230/LIPIcs.FSCD.2020.4. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.33 (cit. on pp. 12, 13, 33).

[14] A. Das and F. Pfenning. "Session Types with Arithmetic Refinements". In: *CoRR* abs/2005.05970 (2020). arXiv: 2005.05970. URL: https://arxiv.org/abs/2005.05970 (cit. on p. 13).

[15] M. Fowler. *Domain-Specific Languages*. URL: https://martinfowler.com/dsl.html (16/2/2022) (cit. on p. 2).

[16] S. Fowler et al. *Exceptional asynchronous session types: session types without tiers*. 2019. URL: https://doi.org/10.1145/3290341 (cit. on p. 13).

[17] D. Griffith. *Polarized Substructural Session Types*. 2015 (cit. on p. 13).

[18] P. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security*. RFC 3207. RFC Editor, Feb. 2002. URL: http://www.rfc-editor.org/rfc/rfc3207.txt (cit. on p. 37).

[19] K. Honda, V. T. Vasconcelos, and M. Kubo. *Language primitives and type discipline for structured communication-based programming*. 1998. URL: https://www.di.fc.ul.pt/~vv/papers/honda.vasconcelos.kubo_language-primitives.pdf (cit. on pp. 2, 5).

[20] K. Honda, N. Yoshida, and M. Carbone. "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by G. C. Necula and P. Wadler. ACM, 2008, pp. 273–284. DOI: 10.1145/1328438.1328472. URL: https://doi.org/10.1145/1328438.1328472 (cit. on pp. 3, 7, 9).

[21] R. Hu and N. Yoshida. *Hybrid Session Verification through Endpoint API Generation*. URL: https://www.doc.ic.ac.uk/research/technicalreports/2015/DTR15-6.pdf (cit. on pp. 15, 17, 24).

[22] R. Hu, N. Yoshida, and K. Honda. *Session-Based Distributed Programming in Java*. 2008. URL: https://dl.acm.org/doi/10.1007/978-3-540-70592-5_22 (cit. on p. 13).

[23] H. Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (Apr. 2016). ISSN: 0360-0300. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052 (cit. on p. 4).

[24] A. Igarashi and N. Kobayashi. "A Generic Type System for the Pi-calculus". In: *Theoretical Computer Science* 311.1 (2004), pp. 121–163. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(03)00325-6. URL: https://www.sciencedirect.com/science/article/pii/S0304397503003256 (cit. on p. 5).

[25] A. Igarashi and N. Kobayashi. "Type Reconstruction for Linear Pi-Calculus with I/O Subtyping". In: *Information and Computation* 161.1 (2000), pp. 1–44. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.2000.2872. URL: https://www.sciencedirect.com/science/article/pii/S0890540100928724 (cit. on p. 5).

[26] K. Imai, N. Yoshida, and S. Yuen. "Session-ocaml: A session-based library with polarities and lenses". In: *Science of Computer Programming* 172 (2019), pp. 135–159. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2018.08.005. URL: https://www.sciencedirect.com/science/article/pii/S0167642318303289 (cit. on pp. 14, 17).

[27] K. Imai et al. "Multiparty Session Programming With Global Protocol Combinators". In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by R. Hirschfeld and T. Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 9:1–9:30. ISBN: 978-3-95977-154-2. DOI: 10.4230

/LIPIcs.ECOOP.2020.9. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13166 (cit. on p. 17).

[28] *Java Microbenchmark Harness*. URL: https://github.com/openjdk/jmh (24/7/2022) (cit. on p. 42).

[29] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. "Session Types for Rust". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 13–22. ISBN: 9781450338103. DOI: 10.1145/2808098.2808100. URL: https://doi.org/10.1145/2808098.2808100 (cit. on p. 17).

[30] *JMH Gradle Plugin*. URL: https://github.com/melix/jmh-gradle-plugin (24/7/2022) (cit. on p. 42).

[31] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. RFC Editor, Oct. 2008. URL: http://www.rfc-editor.org/rfc/rfc5321.txt (cit. on p. 37).

[32] N. Kobayashi. "A Partially Deadlock-Free Typed Process Calculus". In: *ACM Trans. Program. Lang. Syst.* 20.2 (Mar. 1998), pp. 436–482. ISSN: 0164-0925. DOI: 10.1145/276393.278524. URL: https://doi.org/10.1145/276393.278524 (cit. on pp. 4, 5).

[33] N. Kobayashi, S. Saito, and E. Sumii. "An Implicitly-Typed Deadlock-Free Process Calculus". In: *Proceedings of the 11th International Conference on Concurrency Theory*. CONCUR '00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 489–503. ISBN: 3540678972 (cit. on p. 5).

[34] W. Kokke. "Rusty Variation: Deadlock-free Sessions with Failure in Rust". In: *Electronic Proceedings in Theoretical Computer Science* 304 (Sept. 2019), pp. 48–60. ISSN: 2075-2180. DOI: 10.4204/eptcs.304.4. URL: http://dx.doi.org/10.4204/EPTCS.304.4 (cit. on pp. 16, 17).

[35] W. Kokke and O. Dardha. "Deadlock-Free Session Types in Linear Haskell". In: ACM, New York, NY, USA: ACM, 2021. URL: https://doi.org/10.1145/3471874.3472979 (cit. on pp. 14, 17).

[36] *Kotlin Symbol Processing API*. URL: https://github.com/google/ksp/ (16/2/2022) (cit. on p. 46).

[37] *KotlinPoet*. URL: https://square.github.io/kotlinpoet/ (16/2/2022) (cit. on pp. 21, 30).

[38] D. Kouzapas et al. "Typechecking protocols with Mungo and StMungo: A session type toolchain for Java". In: *Science of Computer Programming* 155 (2018). Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016, pp. 52–75. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0167642317302186 (cit. on pp. 12, 14).

[39] *Ktor*. URL: https://ktor.io/ (2/6/2022) (cit. on pp. 31, 43).

[40] N. Lagaillardie, R. Neykova, and N. Yoshida. "Implementing Multiparty Session Types in Rust". In: *Coordination Models and Languages*. Ed. by S. Bliudze and L. Bocchi. Cham: Springer International Publishing, 2020, pp. 127–136. ISBN: 978-3-030-50029-0 (cit. on pp. 16, 17).

[41] S. Lindley and J. G. Morris. "Embedding Session Types in Haskell". In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 133–145. ISBN: 9781450344340. DOI: 10.1145/2976002.2976018. URL: https://doi.org/10.1145/2976002.2976018 (cit. on pp. 14, 17).

[42] *Linear base library*. URL: https://github.com/tweag/linear-base/ (16/6/2022) (cit. on p. 17).

[43] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. iii).

[44] *Mungo*. URL: http://www.dcs.gla.ac.uk/research/mungo/ (16/2/2022) (cit. on p. 14).

[45] R. Neykova et al. "A Session Type Provider: Compile-Time API Generation of Distributed Protocols with Refinements in F#". In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 128–138. ISBN: 9781450356442. DOI: 10.1145/3178372.3179495. URL: https://doi.org/10.1145/3178372.3179495 (cit. on pp. 12, 15, 17, 33).

[46] R. Pucella and J. A. Tov. "Haskell session types with (almost) no class". In: 2008, pp. 25–36. ISBN: 9781605580647. DOI: 10.1145/1411286.1411290 (cit. on pp. 14, 17).

[47] A. Scalas and N. Yoshida. "Lightweight Session Programming in Scala". In: *30th European Conference on Object-Oriented Programming*. LIPIcs. Dagstuhl, 2016, 21:1–21:28. DOI: 10.4230/LIPIcs.ECOOP.2016.21 (cit. on p. 15).

[48] A. Scalas et al. "A linear decomposition of multiparty sessions for safe distributed programming". In: vol. 74. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, June 2017, pp. 241–2431. ISBN: 9783959770354. DOI: 10.4230/LIPIcs.ECOOP.2017.24. URL: https://drops.dagstuhl.de/opus/volltexte/2017/7263/pdf/LIPIcs-ECOOP-2017-24.pdf (cit. on pp. 15, 17).

[49] *Sesh: A library for deadlock-free session-typed communication in Rust*. URL: https://github.com/wenkokke/sesh/ (17/6/2022) (cit. on p. 17).

[50] *sessionkotlin*. URL: https://github.com/d-costa/sessionkotlin/tree/thesis (6/9/2022) (cit. on p. 23).

[51] *sessionkotlin-plugin*. URL: https://github.com/d-costa/sessionkotlin/blob/thesis/sessionkotlin/sessionkotlin-plugin/src/main/kotlin/com/github/d_costa/sessionkotlin/SessionKotlinPlugin.kt (6/9/2022) (cit. on p. 45).

[52] R. Siemborski and A. Melnikov. *SMTP Service Extension for Authentication*. RFC 4954. RFC Editor, July 2007. URL: https://datatracker.ietf.org/doc/html/rfc4954 (cit. on p. 38).

[53] R. E. Strom and S. Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *IEEE Transactions on Software Engineering* SE-12.1 (1986), pp. 157–171. DOI: 10.1109/TSE.1986.6312929 (cit. on p. 12).

[54] *The Z3 Theorem Prover*. URL: https://github.com/Z3Prover/z3 (31/5/2022) (cit. on p. 33).

[55] P. Thiemann and V. T. Vasconcelos. "Context-Free Session Types". In: *SIGPLAN Not.* 51.9 (Sept. 2016), pp. 462–475. ISSN: 0362-1340. DOI: 10.1145/3022670.2951926. URL: https://doi.org/10.1145/3022670.2951926 (cit. on p. 13).

[56] B. Toninho, L. Caires, and F. Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by M. Felleisen and P. Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 350–369. DOI: 10.1007/978-3-642-37036-6_20. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_20 (cit. on p. 13).

[57] B. Toninho and N. Yoshida. "Certifying data in multiparty session types". In: *Journal of Logical and Algebraic Methods in Programming* 90 (2017), pp. 61–83. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2016.11.005. URL: https://www.sciencedirect.com/science/article/pii/S2352220816300864 (cit. on pp. 11, 12).

[58] T. Tu et al. "Understanding Real-World Concurrency Bugs in Go". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 865–878. ISBN: 9781450362405. DOI: 10.1145/3297858.3304069. URL: https://doi.org/10.1145/3297858.3304069 (cit. on p. 1).

[59] N. Yoshida et al. "The Scribble Protocol Language". In: *Trustworthy Global Computing*. Ed. by M. Abadi and A. Lluch Lafuente. Cham: Springer International Publishing, 2014, pp. 22–41. ISBN: 978-3-319-05119-2 (cit. on pp. 12, 15, 29, 36).

[60] F. Zhou et al. “Statically Verified Refinements for Multiparty Protocols”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428216. URL: https://doi.org/10.1145/3428216 (cit. on pp. 11, 12, 17, 32, 33).