NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DAVID MARIA ALMEIDA AMORIM DA COSTA
Bachelor in Computer Science

SESSION KOTLIN

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

DEPARTMENT OF
COMPUTER SCIENCE

SESSION KOTLIN

DAVID MARIA ALMEIDA AMORIM DA COSTA
Bachelor in Computer Science

Adviser: Bernardo Toninho
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 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. [24]

Abstract

Concurrency and distribution have become essential for building modern applications. However, developing and maintaining them is not an easy task. Communication errors are a common source of problems: unexpected messages or mutual dependencies can cause runtime errors and, in the worst case, deadlocks. 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

Contents

List of Figures

List of Listings

Acronyms

CPS	Continuation-Passing Style 13
KSP	Kotlin Symbol Processing 17, 18
RMPST	Refined Multiparty Session Types 19
SMT	Satisfiability Modulo Theory 13, 25

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. 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: 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 would be useful if we could completely avoid these errors 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 when the interactions change over time. 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 [13], 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.

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. Tipically formal 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 [10]. Internal DSLs are languages defined within another language, and are usually 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 pratical to use as they do not require external tools.

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 languaged 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.

Objective 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 [14, 6] 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). This work will use a combination of static (compile-time) and dynamic (runtime) verifications and explore Kotlin's capabilities for DSL development, including type-safe builders, annotation processing, 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 and code generation in the language's framework;
- We provide an implementation that allows for both concurrent and distributed programming.

Outline In Chapter 2 we introduce key concepts to this thesis, namely binary and multiparty session types, discuss existing implementations of session types in other languages, and explore some Kotlin features relevant for implementing a DSL. Chapter 3 describes our work proposal, the challenges that its implementation brings, and the strategies we can use to address them.

2

Background and Related Work

In this chapter, we introduce the concepts that are key to this thesis and discuss related work. In Section 2.1, we begin by presenting the challenges of concurrent applications and how session types help programmers tackle them. We examine binary and multiparty session types, and, in Section 2.2, their existing implementations. In Section 2.3, we argue that Kotlin is a good candidate to embed session types into and present useful metaprogramming tools. Finally, in Section 2.4 we discuss work that is more closely related.

2.1 Session Types

2.1.1 Motivation

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 a protocol that looks like this:

Figure 2.1: An adder server and a client

Server

```
receive an integer
receive an integer
send their sum
```

Client

```
send an integer
send an integer
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. In order to lift this weight off the programmer's shoulders, we can assign a *type* to this *session* (the whole conversation between the client and the server). The compiler, having access to the *session type*, can verify that the operations are performed in an order that everyone expects, thus 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.

2.1.2 Binary Session Types

Session types were first introduced by Honda, Vasconcelos, and Kubo [13], 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.

Figure 2.2: Session types

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

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. Sessions terminate with end.

Figure 2.3: Duality of session types

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

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

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 modulus operations: It offers a choice (&) between `Add`, `Mod`, and `Quit`. The client must have the dual session type: an internal choice (⊕) between the same labels (`Add`, `Mod`, `Quit`). 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 2.4: An arithmetic server and a client

Server

```
Server = &{
  Add: ?Int.?Int!Int.Server,
  Mod: ?Int.!Int.Server,
  Quit: end
}
```

Client

```
Client = ⊕{
  Add: !Int.!Int?Int.Client,
  Mod: !Int.?Int.Client,
  Quit: end
}
```

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

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

Process B

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

Process C

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

These properties are only guaranteed for individual binary sessions: 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.

2.1.3 Multiparty Session Types

Multiparty session types [14, 6] extend the theory behind binary session types to account for more than two processes. Honda et al. [14] 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

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

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 ($G \upharpoonright q$)

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

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

$$end \upharpoonright q = end$$

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. 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

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

Figure 2.9: Local projections of G_1

$$
\begin{aligned}
G_1 \upharpoonright a = \oplus\langle b, \{ & \\
&L_1 : end, \\
&L_2 : ?\langle U, b\rangle.end \\
\}\rangle &
\end{aligned}
$$

$$
\begin{aligned}
G_1 \upharpoonright b = \&\langle a, \{ & \\
&L_1 : !\langle U, c\rangle.end, \\
&L_2 : !\langle U, a\rangle.end \\
\}\rangle &
\end{aligned}
$$

$$G_1 \upharpoonright c = \text{undefined}$$

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. [2] 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 2.10: Global type G_2

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

Figure 2.11: Local projections of G_2

$$
\begin{aligned}
G_2 \restriction a = \oplus\langle b, \{ & \\
& L_1 : end, \\
& L_2 : end \\
\}\rangle &
\end{aligned}
$$

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

$$
\begin{aligned}
G_2 \restriction c = \&\langle b, \{ & \\
& K_1 : end, \\
& K_2 : end \\
\}\rangle &
\end{aligned}
$$

Figure 2.12 describes a two buyer protocol (simplified example from Honda et al. [14]). In this protocol, buyer a starts by sending a message to the seller, s, with the name of the item it wants to buy. Afterwards, s sends the quote to both buyers. With this information, a sends to b the amount of money it can give. Buyer b judges whether a contributed 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.

If we define a well-formed global type for the interleaving of binary sessions detailed 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 2.12: The Two Buyer protocol

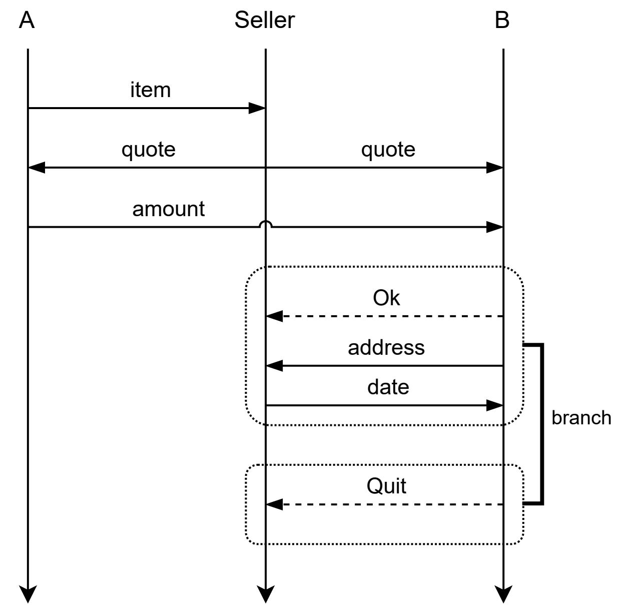


Figure 2.13: Global type $TwoBuyer$

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

Figure 2.14: Local projections of $TwoBuyer$

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

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

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

Figure 2.15: Global type G_3

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

2.2 Implementations of Session Types

Most presentations of session types are developed in formal languages and not general-purpose languages. 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.

2.2.1 Native

Native implementations come in two types: 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 linearity and type evolution.

Toninho et al. [33] 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 [12] explores logically motivated session types and polarization, and presents a language called SILL. Its interpreter is written in Ocaml.

Das and Pfenning [7] 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 showed in their other work [8].

FreeST [1] 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.

Fowler et al. [11] 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 the constructs that the language has. Effect handlers are a generalization of exception handlers.

Additionally, there are some implementations that extend existing languages. Hu, Yoshida and Honda [16] extend Java with session types, with support for delegation subtyping.

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

Mungo [26] 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 [26], a tool that translates local protocols from Scribble to typestate specifications that Mungo can use.

2.2.2 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).

2.2.2.1 Statically checked approach

Fully statically checked implementations are generally very safe, but hard 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. [17]. 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 [28] 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. [23] 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 [19] 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 and Balzer [3] 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.

2.2.2.2 Hybrid approach

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

There are several implementations of session types that take this approach. Hu and Yoshida [15] present `scribble-java`, an implementation in Java for multiparty session types based on API generation as an extension of Scribble. 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 [32] 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 2.1.3) using `scribble-java` [15]. 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 [29] present a library, `lchannels`, that offers an API for binary session programming in Scala with Continuation-Passing Style (CPS) programming. Linearity is enforced during runtime. Scalas et al. [30] build on this work by encoding deadlock-free multiparty sessions as a composition of binary sessions. They also extend Scribble.

Neykova et al. [27] propose a library for the specification and implementation of multiparty distributed protocols in F#. It is implemented by extending and integrating Scribble with a Satisfiability Modulo Theory (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 cannot be verified statically are enforced with assertions during runtime.

In Rust, Lagaillardie et al. [22] present an implementation of multiparty session types as a wrapper of the library for binary session types done by Kokke and Wen [18]. 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

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

```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;
    }
}
```

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.

2.3 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. We describe them below.

2.3.1 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 2.2 shows all the different ways we can invoke map with a lambda: line 4 corresponds to the basic method call everyone is accustomed to using, 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 2.2: 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 2.3 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 2.3: An extension function

```
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 2.4: Function literal with receiver

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

Listing 2.4 declares a function literal with receiver that has the same functionality as the extension function declared in Listing 2.3.

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

2.3.2 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 2.3.1).

Figure 2.16 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 and 19 generate two list items, showing that we can write any code we want inside the lambdas.

Figure 2.16: Kotlin DSL for HTML (Simplified example from the kotlin docs)

HTML definition

```
 1  val result = 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

```
<html>
  <head>
    <title>
      HTML encoding with Kotlin
    </title>
  </head>
  <body>
    <h1>
      HTML encoding with Kotlin
    </h1>
    <p>
      an alternative markup to HTML
    </p>
    <a href="http://kotlinlang.org">
      Kotlin
    </a>
    <p>
      some text
      <ul>
        <li>
          1*2 = 2
        </li>
        <li>
          2*2 = 4
        </li>
      </ul>
    </p>
  </body>
</html>
```

2.3.3 Annotation Processing

Kotlin Symbol Processing (KSP)[20] is an API to process Kotlin programs idiomatically, that allows us to visit classes, class members, functions and their parameters.

Supports incremental and multiple round processing, but cannot examine expressions or statements and cannot modify code. KSP is designed to hide compiler changes and is designed to not be tied to the JVM.

As an example, let's define an annotation that extend a class with a function that prints every property of the class.

```
@Target(AnnotationTarget.CLASS)
annotation class PrintAll(val prefix: String)
```

We can now create a class, add the annotation, provide a value for the `prefix` argument, and call the generated function.

```
@PrintAll(prefix = "--> ")
class MyClass (
  val arg1: String,
  val arg2: List<Int>,
  val arg3: List<Map<String, *>>,
)

fun main() {
  val obj = MyClass(
    "hello",
    listOf(1, 2),
    listOf(mapOf(Pair("hello", true)))
  )
  obj.printAll()
}
```

```
--> hello
--> [1, 2]
--> [{hello=true}]
```

To process the annotation, we need to implement a `SymbolProcessor` and a `Visitor`. The `SymbolProcessor` is responsible for validating the annotated symbols (for example, to make sure that it is a class declaration) and for instantiating the visitor for each one of them. The `Visitor` is a class that contains methods that define what must be done for each kind of symbol: classes, functions, properties, etc.

2.3.4 Metaprogramming

KotlinPoet [21] 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. It also has an extension for converting KSP types to KotlinPoet types and writing to KSP's `CodeGenerator`, which is the class responsible for creating files.

In Figure 2.17 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 through 10, define the class. Lines 4 to 6 define the primary constructor with an argument `name`, lines 7 to 9 define a property `name` that is initialized with the value of the argument.

Figure 2.17: 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
)
```

2.3.5 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 2.18 shows how we can build programs with them. In line 2, we use the `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`.

2.3.6 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 when receiving if the buffer is empty. It is possible to define a channel with an "unlimited"

Figure 2.18: 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!
```

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 2.18 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.

2.4 Related work

Zhou et al. [34] propose a deadlock-free implementation of Refined Multiparty Session Types (RMPST), extending Scribble to target F*, a verification-oriented functional programming language. The main difference from other implementations is that session I/O is not written directly: the developer only writes callback functions that are invoked by a runtime. With this strategy, session linearity is achieved by construction, without a linear type system or runtime checks.

Some interesting implementations of DSLs in Kotlin include a Prolog DSL [4] and a DSL for differential programming, Kotlin∇ [5]. Both solutions make use of Kotlin's support for operator overloading [2], lambda expressions, functions with receiver, and extension methods.

One topic that is close to the essence of session types is typestates. Typestates can be roughly described as session types with a *"server"* (the API) and a *"client"* (the API client). They allow us to define an API as a finite state machine and have its transitions enforced by the type system. Duarte and Ravara [9] developed a DSL to encode typestates in Rust. The annotated code is parsed and a state machine is built from the specification; if it passes all verifications, code is generated. Mota et al. extended the Checker Framework with a plugin for typestates, JaTyC [25]. The Checker Framework [31] enhances Java's

[2] https://kotlinlang.org/docs/operator-overloading.html

type system by letting developers define restrictions like variable nullness, must-call methods before an object is dropped, variables that should not be aliased, among others. JaTyC statically checks that class methods are invoked in the correct order, specified in a file. This tool provides better error messages than Mungo, for some cases (nullness and linearity checking, protocol completion), and presents extra features. A full comparison can be found on their GitHub wiki.

3 Proposed Work

We propose an implementation of Multiparty Session Types in Kotlin that is idiomatic and explores the features of the language. We propose a DSL, built upon the features mentioned in Section 2.3, that enables programmers to declare and use session types in a practical way. We intend to statically guarantee error-free communication. If we use a callback-based approach, we can guarantee deadlock-freedom and linearity by construction; using a fluent API, we have a hybrid implementation with runtime linearity checks. These programming styles are described below, in Subsection 3.1.2.

In Section 3.1, we discuss the implementation challenges that the above proposal brings. In Section 3.2, the work plan is presented.

3.1 Implementation Challenges

3.1.1 Representing Session Types in Kotlin

The first question we need to answer is how to represent session types. As discussed in Section 2.1, MPST have two layers: the global type, that describes the *"choreography"* of the interactions, and the local types, that describe the actions pertinent to each role.

First, we need some way to represent global types. For this, we could go in two different paths: either define them using Scribble and extend the Scribble toolchain to target Kotlin (similar to Hu and Yoshida [15]), or define them directly in Kotlin, using a DSL built with type-safe builders described in Subsection 2.3.2. Afterward, we must validate the global protocol, ensuring its projection is defined for all participants, as defined in Subsection 2.1.3.

If the global type is defined in Kotlin, we have two options: create some hook to Scribble and use it to validate, or implement validation ourselves inside the Kotlin environment (probably using an annotation). It is necessary to investigate whether these options are feasible, and what is the best for our case.

Lastly, the local types should be projected either using Scribble or by implementing our projection function, with a combination of annotation processing (Subsection 2.3.3) and code generation (Subsection 2.3.4).

Listing 3.1: Possible definition of the Two Buyer protocol in Kotlin

```kotlin
val twoBuyerProtocol =
  globalProtocol(A, B, S) {
    send(A, S, String::class.java)
    send(S, A, Int::class.java)
    send(S, B, Int::class.java)
    send(A, B, Int::class.java)
    branch(B, S) {
      case("Ok") {
        send(B, S, String::class.java)
        send(S, B, String::class.java)
      }
      case("Quit") {
        end()
      }
    }
  }

@ProjectLocal
class TwoBuyerS : LocalProtocol(twoBuyerProtocol, S)
```

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

Listing 3.1 illustrates a possible definition of the global type for the same Two Buyer protocol in Kotlin, making use of type-safe builders (Subsection 2.3.2). After defining the global protocol (`twoBuyerProtocol`), we could create a local API (`TwoBuyerS`) by extending some `LocalProtocol` class, passing the global protocol and the role to be projected onto as arguments. By tagging the class with `@ProjectLocal`, extension methods and any other classes needed for the protocol are generated by the annotation processor.

3.1.2 Programming Style

The next step is to define how local types map to usable code. We explore two strategies: generate a fluent API [15] or take a callback-based approach [34].

3.1.2.1 Fluent API

A fluent API consists of generating a sequence of classes, one for each state of the protocol. Each class has methods for every action available for the corresponding state. Only the class that corresponds to the initial state has a public constructor; objects of the other classes are returned when performing actions, such as receiving and sending messages.

The programmer instantiates an object of the initial class, calls one of its methods, and the return value is an object of the class of the following state. Usually, linearity is enforced during runtime and is guaranteed by including guards on these methods, to allow only a single call. This architecture allows the programmer to chain method calls, making it *fluent*. Listing 2.1 is an example of such implementation.

Listing 3.2 shows how we could use the `TwoBuyerS` class specified in Listing 3.1 to implement the seller's endpoint in this style, using type-safe builders (Subsection 2.3.2). The `Buffer` class is a generic class that has a public field. The `when` expression, similar to Java's `switch`, ensures that all branches of the choice are handled.

Listing 3.2: Possible seller endpoint implementation of the Two Buyer protocol in Kotlin

```
val stringBuffer = Buffer("")

TwoBuyerS()
  .receive(stringBuffer)
  .send(100)
  .send(100)
  .branch {
      when (choice) {
          Ok ->
            receive(stringBuffer)
              .send(Date.from(Instant.now()))

          Quit -> {}
      }
  }
```

3.1.2.2 Callbacks

The idea of a callback-based approach is to invert the control and have the library call user-defined code. Inspired by the work by Zhou et al. [34], we could have a thread, not available to the programmer, perform the communication and invoke user-defined functions (the *callbacks*) to handle messages 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 exceptions that the API approach cannot escape from. On the other hand, the programmer needs to code in an event-driven fashion.

Listing 3.3: Callbacks for the Seller from the Two Buyer protocol (F*)

```
let callbacks : callbacksS = {
  (* state32OnreceiveBookTitle : (st: state32) -> (s: string) -> ML (unit); *)
  state32OnreceiveBookId = (fun _ id ->
    FStar.IO.print_string "S: Received book id: ";
    print_int id;
    FStar.IO.print_string "\n"
  );

  (* state34OnsendQuoteA : (st: state34) -> ML (int); *)
  state34OnsendQuoteA = (fun _ -> 100);

  (* state35OnsendQuoteB : (st: state35) -> ML (int); *)
  state35OnsendQuoteB = (fun _ -> 100);

  (* state36OnreceiveOk : (st: state36) -> (address: int) -> ML (unit); *)
  state36OnreceiveOk = (fun _ _ -> FStar.IO.print_string "S: Received Buy\n");

  (* state36OnreceiveQuit : (st: state36) -> (_dummy: unit) -> ML (unit); *)
  state36OnreceiveQuit = (fun _ _ -> FStar.IO.print_string "S: Received Cancel\n");

  (* state37OnsendDelivery : (st: state37) -> ML (int); *)
  state37OnsendDelivery = (fun _ -> 1643646039)
}
```

Listing 3.3 shows the callbacks that the programmer needs to define to implement the seller endpoint from the Two Buyer protocol (Figure 2.12), using the toolchain proposed by Zhou et al. [34] in F*.

Function `state32OnreceiveBookId` defines what to do when receiving a request for a book; functions `state34OnsendQuoteA` and `state35OnsendQuoteB` define what the seller sends to the buyers as the quote. Functions `state36OnreceiveOk` and `state36OnreceiveQuit` define what the seller does when receiving the choice from *b*. The last function, `state37OnsendDelivery`, is for sending the delivery date to *b*.

3.1.3 Communication Backend

The third step is how to determine the communication backend. The first decision is between an implementation that is simply *concurrent*, or *distributed*. To support concurrency, we can use kotlin channels; for distribution, sockets.

Channels preserve message order, as well as (TCP) sockets, for the same sender. This makes them suitable options for communication between session peers. Distributed architectures have the downside of being susceptible to network errors (timeouts, for example). We will not be addressing this kind of errors.

The best option, and the one we would like to choose, is to leave this decision for the programmer, and support both channels *and* sockets. Figure 2.18 illustrates how we can build concurrent applications using Kotlin's coroutines and channels.

If the fluent API is implemented, we can offer different constructors for each communication scheme; if the callback method is used, the user needs to provide functions that send and receive values to the other roles, similarly to Zhou et al. [34].

3.1.4 Safety Guarantees

If the fluent API implementation is selected, protocol fidelity is statically guaranteed if channels are used linearly (i.e. each state is only used once). If channel linearity is not respected, runtime assertions will prevent the same protocol state from being used multiple times. In any case, no endpoint should receive unexpected messages. The only circumstance in which deadlock-freedom cannot be guaranteed is if the connection order between endpoints is not described explicitly in the global protocol. For example, if two endpoints are waiting for a connection (both "behaving" as a server), they will not progress.

If the callback implementation is used, channel linearity is guaranteed by construction as a result of I/O not being called directly by the developer. Protocol fidelity and deadlock-freedom should be statically guaranteed.

In distributed environments, the transport layer is expected to be reliable: it should not fail or drop messages.

3.1.5 Possible Extensions

The first extension that comes to mind is type refinements. If we extend our syntax for MPST and name the message values, we can define logical expressions to constrain the exchanged data.

Figure 3.1: A refined global type

$$
\begin{array}{l}
TwoBuyerRefined \triangleq \\
\qquad a \to s : \langle \text{title} : String \rangle . \\
\qquad s \to a : \langle \text{quoteA} : Int \rangle . \\
\qquad s \to b : \langle \text{quoteB} : Int\ [quoteB = quoteA] \rangle . \\
\qquad a \to b : \langle \text{aShare} : Int\ [aShare \geq quoteA/2] \rangle . \\
\qquad b \to s : \{ \\
\qquad Ok : \\
\qquad\qquad b \to s : \langle \text{bShare} : Int\ [aShare + bShare = quoteB] \rangle . end, \\
\qquad\quad Quit : end \\
\qquad \}
\end{array}
$$

We show, in Figure 3.1, how we could extend the global type of the Two Buyer protocol, originally defined in Figure 2.13, to include value refinements. These refinements, defined between brackets, ensure that the quote sent to a is equal to the one sent to b ($quoteB = quoteA$), that a's share is at least equal to half of the quote ($aShare \geq quoteA/2$), and that the sum of what a and b offer are equal to the quote ($aShare + bShare = quoteB$).

The first refinement can be guaranteed statically by generating the corresponding method without arguments and using the same value sent in the last message. The other two cannot be verified statically because they depend on user input: we need to include assertions that verify them during runtime.

Refinements need to be validated to ensure they are *satisfiable*. This can be done by employing an external solver (for example, an SMT solver) [27, 34, 7]. The solver needs to verify both *logical* satisfiability and name visibility (whether the expressions reference available information).

3.2 Work Plan

In this section we delineate the steps needed to take to implement the proposition defined in this chapter.

Task 1 - Global type representation: Compare the possibility of extending Scribble with creating a DSL in Kotlin, to represent multiparty session types. Explore Kotlin's DSL capabilities with type-safe builders.

Task 2 - Global type validation: Analyse whether using Scribble to validate the global protocol is feasible and compare it with native validation inside Kotlin's ecosystem.

Task 3 - Local projections: Study the possibility of using Scribble to project the global type vs implementing it in Kotlin, using a combination of annotation processing and code generation techniques.

Task 4 - Mapping local types to code: Explore the two possibilities for local code: fluent API with chained method calls, and a callback-based approach. Analyse the advantages and disadvantages of each, keeping in mind the ease-of-use and the guarantees we want to provide to programmers.

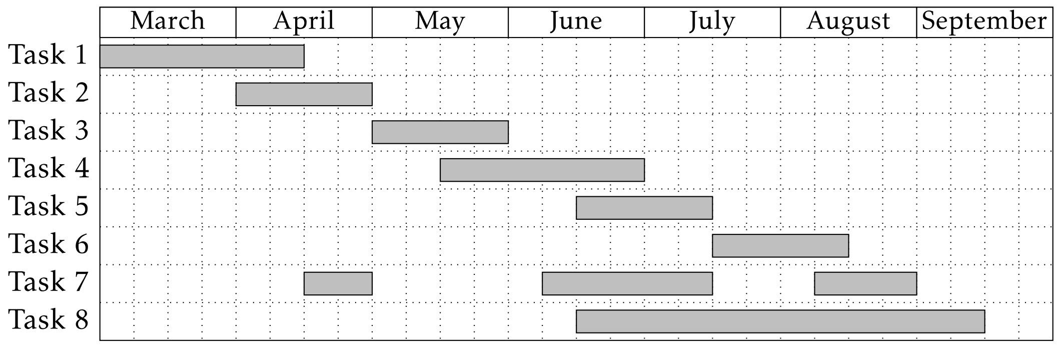
Task 5 - Communication backend: Consider the possibility of implementing the backend in a way that it supports concurrent applications, using channels (Subsection 2.3.6), and distributed applications, using sockets.

Task 6 - Extensions: Contemplate possible extensions, namely type refinements.

Task 7 - Validation / Evaluation: Write comprehensive use-cases that show that the implementation is valid, expressive and useful, in both concurrent and distributed applications.

Task 8 - Document: Compose the final document.

Figure 3.2: Gantt Chart for the proposed work



Bibliography

[1] 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. 11).

[2] 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. 8).

[3] R. Chen and S. Balzer. "Ferrite: A Judgmental Embedding of Session Types in Rust". In: ACM, New York, NY, USA: ACM, 2021. URL: https://arxiv.org/abs/2009.13619 (cit. on p. 12).

[4] G. Ciatto et al. "2P-Kt: logic programming with objects & functions in Kotlin". In: *Proceedings of the Workshop on 21st Workshop "From Objects to Agents", Bologna, Italy, September 14-16, 2020*. Ed. by R. Calegari et al. Vol. 2706. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 219–236. URL: http://ceur-ws.org/Vol-2706/paper14.pdf (cit. on p. 19).

[5] B. Considine. *Programming Tools for Intelligent Systems*. 2020. URL: https://papyrus.bib.umontreal.ca/xmlui/bitstream/handle/1866/24310/Considine_Breandan_2020_Memoire.pdf (cit. on p. 19).

[6] 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).

[7] 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. 11, 25).

[8] 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. 11).

[9] J. Duarte and A. Ravara. "Retrofitting Typestates into Rust". In: *25th Brazilian Symposium on Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 83–91. ISBN: 9781450390620. URL: https://doi.org/10.1145/3475061.3475082 (cit. on p. 19).

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

[11] S. Fowler et al. *Exceptional asynchronous session types: session types without tiers*. 2019. URL: https://doi.org/10.1145/3290341 (cit. on p. 11).

[12] D. Griffith. *Polarized Substructural Session Types*. 2015 (cit. on p. 11).

[13] 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).

[14] 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).

[15] 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. 13, 21, 22).

[16] 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. 11).

[17] 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 p. 12).

[18] 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 p. 13).

[19] 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 p. 12).

[20] *Kotlin Symbol Processing API*. URL: https://github.com/google/ksp/ (16/2/2022) (cit. on p. 17).

[21] *KotlinPoet*. URL: https://square.github.io/kotlinpoet/ (16/2/2022) (cit. on p. 17).

[22] 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 p. 13).

[23] 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 p. 12).

[24] 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).

[25] J. Mota, M. Giunti, and A. Ravara. "Java Typestate Checker". In: *Coordination Models and Languages*. Ed. by F. Damiani and O. Dardha. Cham: Springer International Publishing, 2021, pp. 121–133. ISBN: 978-3-030-78142-2. URL: https://github.com/jdmota/java-typestate-checker (cit. on p. 19).

[26] *Mungo*. URL: http://www.dcs.gla.ac.uk/research/mungo/ (16/2/2022) (cit. on p. 11).

[27] 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. 13, 25).

[28] 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 p. 12).

[29] 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. 13).

[30] 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 p. 13).

[31] *The Checker Framework*. URL: https://checkerframework.org (16/2/2022) (cit. on p. 19).

[32] *The Scribble language*. URL: http://www.scribble.org/ (16/2/2022) (cit. on p. 13).

[33] 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. 11).

[34] 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. 19, 22–25).