Implementing type-checked communication protocols in Python using SessionPy

Martini, Frederik Henrik   Rasmussen, Johan Irvall

{frem , jirr} @ itu.dk

June 2022

Abstract

The increasing importance of communication and coordination between distributed systems presents today's software developers with new challenges, when it comes to ensuring communication happens following some well-formed protocols that guarantee the absence of deadlocks. Session types, along with the protocol description language Scribble, provide the mechanisms for modelling communication between two or multiple parties and ensuring the correctness of the communication sequence. Previous work on embedding these ideas into mainstream dynamic languages has been relying on dynamic verification that can only catch errors at runtime. In this paper, we first lay out the recent groundwork and extensions done on the Python type system. We then present our work on integrating session types into the Python language with a recursive syntactical definition that allows the representation of core session type operations such as message passing, branching, and recursion. The type system in combination with the implemented session types is capable of modelling a series of interactions between two or more parties. These parties can verify their usage of endpoints in a just-in-time stage where any violations of the specified protocol are caught. This project demonstrates a gradual type-checker implemented in Python and presents an endpoint abstraction for working with session types that supports inter-process communication using Python's low-level networking interface; the socket.

Acknowledgement

We would like to thank our supervisor Marco Carbone for months of guidance and insight into the world of session types, and also Alessandro Bruni for his counselling and expertise in advanced Python usage that was a real eye-opener in the early stages.

Contents

1 Introduction

In today's world of software development, communication and coordination between computers is becoming ever more significant. With web services and distributed computing rising in popularity, ensuring their validity becomes an increasingly important task. At present time, around 11 billion devices are connected to the internet and are presumed to increase to 25 billion by 2030 [1]. This increases the pressure on developing tools that allow for correct and effective communication in distributed systems. These systems are by nature difficult to design and implement since the responsibility that a given communication protocol is used correctly, is placed on the individual participants. This results in increasingly complex programs where the low-level communication primitives obfuscate program flow and make it difficult to verify, reason about, and potentially unsafe to execute. The most common issues encountered when writing communication programs are synchronisation issues that can arise when a programmer fails to handle specific incoming messages, or send messages at the proper timing, which may cause a deadlock. A potential remedy to this problem are session types. Session types provide a type-centric approach for explicitly describing both concurrent and distributed systems along with the causal relationship between the communicating peers. As an example, the following local session type describes a simple protocol between a client and a server from the server's point of view. The server must first receive a string from the client and thereafter choose to either send an integer back or end the protocol.

$$?\mathbf{str}.\oplus\{ok : !\mathbf{int}.\mathbf{end},\ ko : \mathbf{end}\} \tag{1}$$

In the context of session types, receive ($?\mathbf{t}$) denotes an input of type $\mathbf{t}$ with its opposite (dual) operation ($!\mathbf{t}$) outputting some type $\mathbf{t}$. Offer ($\&$) provides n labelled session types to choose from, and its dual operation, choose ($\oplus$), picks a choice among these n labelled options. Such an explicit representation of a communication sequence helps solve the previously mentioned synchronisation issues and provides a clear picture of what a communication protocol entails. The following properties are guaranteed by session type as presented by [5]:

1. Interactions within a session never incur a communication error (communication safety).
2. Channels for a session are used linearly (linearity) and are deadlock-free in a single session (progress).
3. The communication sequence in a session follows the scenario declared in the session type (session fidelity, predictability)

Contrary to the previously specified local type, a global type captures all the interactions in some session type. It serves as the basis for the projection onto individual local types that governs the series of interactions the individual must perform to adhere to the session type. These protocols can be written in the protocol-description language Scribble, which provides a means of describing communication protocols with several participants in a systematic, clear way.

Python is a programming language popular for writing distributed systems and web applications, and due to its dynamic type system, current implementations for verifying session types involve doing runtime monitoring and verification as described in [8]. As a result of this, the verification is only available during the execution of protocols making it impossible to catch any session type errors before runtime. This is especially undesirable when systems scale and thorough code-analysis is required. It was found that this issue calls for a well-suited implementation of session types in dynamic languages that can provide programmers with additional tools to verify their communication protocols.

This paper introduces SessionPy, a high-level implementation of session types written in Python with a focus on both static and dynamic analysis that seeks to provide the programmer with additional information regarding their use of session types. SessionPy hides the low-level implementation of communication primitives, and enforces protocols using session types. Unlike previous implementations, SessionPy does not rely exclusively on dynamic monitoring, but instead on a gradual approach that combines static checks and dynamic validation of programs. When type systems allow for both, it is popularly called gradual typing and attempts to combine the robustness of static type systems with the flexibility of dynamic type systems. In this implementation, it is achieved by exploiting built-in functionality for incorporating static analysis by inspecting Python's abstract syntax tree (AST) in combination with decorators and type hints added to Python in the Python Enhancement Proposal (PEP) 484. This rejects ill-typed programs before executing them and informs the programmer of errors in their program statically. The implementation proved to be able to describe selected distributed systems in a Scribble-like syntax and create a functioning Python implementation for each role using the Endpoint API, that lives up to the guarantees of session types.

Section 2 provides an overview of static, dynamic, and gradual type systems found in today's programming languages alongside an overview of Python's type system with its recent additions followed by the introduction of session type. Fundamental language engineering concepts are then explained and Scribble is presented. Section 3 describes the formal rules of the implemented type systems followed by the Python implementation of the type checker, session types, endpoints, and projector. Section 4 presents the results of using SessionPy to model different distributed systems and discusses the implementation and related work. Section 5 finally concludes the project.

2 Background

This section will first present some of the fundamental typing disciplines found in today's programming languages in 2.1. This is followed by a walk-through of the current state of the Python type system in section 2.2 where recently added language features in Python are discussed that make it possible to enforce type hints for primitives and functions. This includes a description of the metaprogramming aspect in Python using abstract syntax tree (AST) inspection and type hints that can be used to write a gradual type system. The basics of session types is explained in section 2.3 where both binary, multiparty and gradual session types are covered. Basic language engineering theory will be presented in section 2.4 lastly followed by a presentation of Scribble in section 2.5.

2.1 Type Systems and Paradigms

Generally, a *type* is a way of describing a set of values that may have operations in common. Most programming languages ship with primitive types for describing integers, floating-point values, string literals, and booleans that all commonly have a concise, syntactic representation (these are respectively `int`, `float`, and `bool` in Python). Customarily, the programmer has options to extend this system of types with classes (in object-oriented languages like Java, C#), or using union types (in functional languages like Haskell, F#). Types often have operations for combining them into fresh values of the same, or even other, well-defined types. The aim of types can be several, also depending on what type system it exists within, but are typically to enhance the *readability* of code; bridging the syntax to natural (English) language. Also commonly, and even more critically, types can help detect errors up-front when they are combined in unexpected ways. A type error is defined as trying to unify, or in some way combine, two values where this is not possible. A trivial example could be adding an integer and a string in languages that do not support implicit conversions, or invoking a method on an object that has not defined said method. Benjamin Pierce's definition of types is especially concerned with software's behaviour: *"A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute"* [7].

This section provides a brief overview of the different typing disciplines found in modern programming languages; static, dynamic, and gradual typing.

2.1.1 Static and Dynamic Typing

Static and dynamic type systems are by far the most dominant typing paradigms found in most programming languages. The two differ primarily in terms of *when* and *how* the program text is analysed.

Static typing is the analysis of code with a focus on classifying types and checking for errors that do not require running the program. That is, what can be known from a semantic analysis of the raw text. When compiling programs in statically typed languages, like C or Java, this typing discipline is enforced on the program before the generation of bytecode. Using explicit keywords (as in C, Java) or by type inference (in OCaml, Python), values are associated with a type during static analysis. Once a variable has been declared it can never change type in its current scope. A key distinction from dynamic typing is that every single line of code gets analysed, even though a certain program path *could* be ruled out. The following code snippet does not type check even though the else branch would never be executed. This is because the compiler takes a conservative approach and reports anything that might result in a type error, instead of trying to figure out which branch of an if-then-else statement is executed.

```
if (true) {
    addOne(42);
} else {
    addOne("wrong type");  // compilation error
}
```

Dynamic typing opposite static typing, performs the checking of types during *runtime*. The compiler, or interpreter, will not do any checks in advance (other than sometimes checking the well-formedness of the program). Dynamic type systems found in languages like Python, JavaScript or Ruby, are generally more flexible to use and can allow for faster development speed as languages using dynamic typing are often interpreted, and do not report all possible type errors. In contrast to static typing, variables in dynamically typed languages are allowed to change type during program execution. Since all paths of the code do not get checked statically, the previous example will type-check in Python with dynamic typing as the else branch might as well not exist at all:

```
if True:
    addOne(42)
else:
    addOne('wrong type') # TypeError, but is never run
```

2.1.2 Gradual Typing

Gradual typing is a hybrid approach combining static and dynamic typing that allows for some parts of a program to be statically checked, and others dynamically[1]. This behaviour can be directed in dynamic languages by using type annotations (TypeScript), and in static languages by explicitly declaring a type as dynamic (Dart, C#). The annotated variables are statically checked while the rest is dynamically checked. Unannotated, or dynamically declared variables are handled by giving them a placeholder *unknown* type (**?**), also known as the dynamic type, that allows implicit casting to any other type, and vice-versa. Gradual types allow dynamic languages to become more static to increase performance and maintainability of for example legacy systems, or allow for said systems to be integrated with new, dynamically typed modules. The approach of dynamic-first is often seen as applications are initially built to get a prototype working fast but often results in large codebases that are hard to scale and maintain as the application grows for reasons described in the beginning of the section. From a software engineering perspective, gradual type systems can be helpful, as it allows the initial application to be developed quickly, and reinforced with types later to increase robustness. Dart[2] is one example of a language with gradual typing—the following example demonstrates the gradual aspect of the language:

```
if (true) {
    print("hello, world!");
} else {
    dynamic x = 2;
    print("hello" + x); // RuntimeError: type 'int' is not a subtype of type
     ↪  'String', but is never run
}
```

Dart will first attempt to statically verify the program, in this case attempting to add an integer to a string results in a type error and the program cannot be run. However, assigning x to a variable with the *dynamic* type essentially disables static checking for this variable. This allows the program to run, but will result in an error at runtime if the *else* branch is run as the *plus* operator for a string and integer is not defined. This works, as gradual types can be implicitly converted when two types are said to be *consistent*, denoted with S $\sim$ T when types S and T are consistent and S !$\sim$ T when they are not [3]:

[1] https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/
[2] https://dart.dev

1. $\forall T \mid ? \sim T \wedge T \sim ?$
2. $\forall B \subset \{int, str, ...\} \mid B \sim B$
3. $\forall \text{fun}(T_1, ..., T_n, T_{ret}), \text{fun}(S_1, ..., S_n, S_{ret}) \mid T_1 \sim S_1 \wedge ... \wedge T_n \sim S_n \wedge T_{ret} \sim S_{ret}$

The first rule states that the dynamic type is consistent with any type T and vice-versa, allowing up- and down-casting. Rule 2 says that any basic type is only consistent with itself and the last rule says that function arguments and its return type should be consistent with the called functions parameters and its return type. In short, the three typing paradigms can be seen as ranging from static typing with a more strict type system, to dynamic typing where types are more relaxed, and finally gradual type systems falling somewhere in between (figure 1).

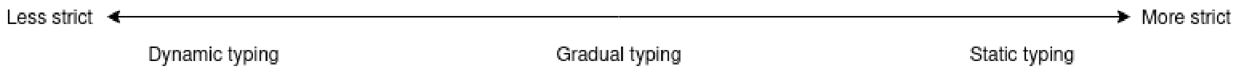


Figure 1: Type strictness

2.2 Python Type System

Python is a dynamically, duck-typed language that does not enforce any type checking upon primitives or functions prior to execution. Duck-typing is a concept related to dynamic typing and refers to the fact the type of an object is less important than the methods it provides. Recently, a series of Python Enhancement Proposals (PEPs) [2] have been suggested and implemented, strengthening Python's capabilities for implementing advanced types with new annotations, type variables, and generic constructs. To verify the usage of session types before the execution of a program, the recent type additions for Python are combined with its meta-programming aspects to achieve static type verification of session types. This section will expand on the data model in Python, how types are defined and what the recent proposals entail. From here on, whenever Python, or *current Python* is referred to, it refers to Python 3.10[3].

2.2.1 Types in Python

Python does not have an explicitly written type system as seen in functional languages like Haskell or OCaml but rather relies on class definitions like Java or C#. Instantiating these classes creates *objects*, and is what makes up all data in the language, and really, all of Python: *"Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects."*[4]. This fact is true even for built-in data types; annotating something to be of integer type as in `x: int = 5` will annotate the variable with the concrete class definition of integer found in the standard library. Sticking with integers as a motivational example, this adds the flexibility to instantiate them without arguments (the integer zero), or with arguments for explicit type conversion as in: `int(3.14) == 3`. User-defined classes are also immediately available as types for this reason—see listing 1 to see the syntax for class (and type) creation of `A`, and how `B` can be subtyped.

[3] https://www.python.org/downloads/release/python-3100/
[4] https://docs.python.org/3/reference/datamodel.html

Listing 1 Minimal example of class creation and subtyping

```python
class A: ...
class B(A): ...
b = B()
isinstance(b, A)   # True
```

The `type` metaclass is an integral part of how classes and objects are related in Python. Everything being an object, basic types like `int`, `str` and `float` are objects too, and the types of these objects are all `type` which refers to the metaclass. Class definitions in Python using `class` keyword are syntactic sugar of a call to `type(name, bases, members)` that by providing the expected arguments name of the class, classes it should inherit from, and member (method) definitions, creates a new class. To demonstrate this brief, but non-trivial fact, consider how the classes (and types) `A` and `B` from listing 1 are defined using this metaclass below.

```
> A = type("A", (), {})
> B = type("B", (A,), {})
> b = B()
> isinstance(b, A)
True
> type(A)
type
> type(int)
type
> type(type)
type
```

This snippet especially highlights how `type` are the underlying class behind all classes and objects in Python.

A strongly typed language like Python implies that it does not do implicit conversion of types (type coercion), but it does allow for certain sets of types to be reduced to larger types of the set if the operations used between them are well-defined. To highlight its strongly typed nature, consider Java, a renowned statically typed language, being even more tolerant than Python when working with string conversion (due to underlying `Object` having defined a `toString` method), making `String pi = "Pi is " + 3.14;` in Java sound, but the similar expression in Python is ill-typed and rejected during runtime. However, there are several examples of well-defined operations between different types, like integers and floating-point values that get lifted into a floating-point type, retaining most information possible. A less common example found in Python is the operations between integers and lists, such as the well-typed expression `3 * [2] + [1]` resulting in `[2, 2, 2, 1]`. In Python, the multiplication of integers and lists is well-defined, yielding a new list of repeated elements which here is then appended using the plus operator to an existing list. This has nothing to do with underlying coercion since none of these values changed their type implicitly during execution.

2.2.2 PEP 484

Type Hints can be thought of as typing *make-up*: a way to enrich data structures, functions or variables with type information by annotating them. These annotations have no effect if no external type checker are used—like mypy[5], pyre[6] or pyright[7]—but will then merely be there to help programmers in reading the otherwise type-lacking code. To showcase the type-negligent nature of type hints, consider the identity function that is annotated to accept a string, and return a string:

```
def string_identity(s: str) -> str:
    return s

> string_identity("hello world")
'hello world'
> string_identity([1,2,3])
[1,2,3]
> string_identity(42)
42
```

None of the above function calls results in any errors, nor produces any warnings. Likewise the following example would still be valid when using type hints even though x is hinted at being an integer, it is still possible to pass variables of other types to the function. This function, unlike `string_identity`, results in an error if a string is passed as an argument since adding strings and integers in Python are not well-defined:

```
def add_one(x: int) -> int:
    return x + 1

> add_one(41)
42
> add_one("foo")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

After the definition of `add_one`, the function itself carries a dictionary containing the type information which can be retrieved by calling `__annotations__` on the function. This dictionary maps variables to the types which they are hinted at providing enough information to write an external type checker to catch these errors up-front.

```
> add_one.__annotations__
{'x': int, 'return': int}
```

TypeVar and Generic make it possible to add type parameters to variables or functions with user-defined names. One usage is to highlight the generic nature of a function:

```
A = TypeVar('A')
def identity(a : A) -> A:
    return a
```

Another usage is in combination with the generic type constructor that makes it possible for an object to have multiple generic types or even recursive types. A class that describes a key-value mapping could be defined like so:

```
K = TypeVar('K')
V = TypeVar('V')
class Mapper(Generic[K,V]):
```

[5]`http://mypy-lang.org`
[6]`https://pyre-check.org`
[7]`https://github.com/microsoft/pyright`

```
    ...
    def get(k : K) -> V:
        ...
        return v
```

Or consider the esoteric example of writing binary numbers using a recursive type definition.

```
One, Zero = TypeVar('One'), TypeVar('Zero')
A, B = TypeVar('A'), TypeVar('B')
class Binary(Generic[A,B]):
    ...

> num101 = Binary[One, Binary[Zero, One]]()
```

Parameterised types can be written using both upper and lowercase syntax, due to legacy and frequent updates of Python. Parameterised generics like `List[int]` and `list[int]` for the case of a list of integers are respectively of type `typing._GenericAlias` and `types.GenericAlias`, and are both valid to use as type annotations in current Python.

2.2.3 Decorators

A *decorator* is a function that returns a function and can be annotated above functions, or even class definitions. With decorators, you can do meta-programming by adding functionality to existing functions that allow arbitrary Python code to be run before a function is initialised or executed while taking the function into account. To motivate how to exploit this for type checking purposes, the identity function is revisited, and its parameter is changed to be an integer making it ill-typed:

```
def string_identity(s : int) -> str:
    return s

> string_identity(42)
42
```

As can be seen and as discussed in section 2.2.2, Python will happily accept and run this program despite it breaking its type-level definition. Using decorators in combination with the type hints provided, the function definition can be rejected before being called:

```
def enforce_identity(f):
    anns = f.__annotations__
    ret_typ = anns.pop('return')
    in_typ = list(anns.values())[0]
    if in_typ != ret_typ:
        raise TypeError(f'{in_typ} was not the same as {ret_typ}')

>> @enforce_identity
.. def string_identity(s : int) -> str:
..     return s

'TypeError: <class 'int'> was not the same as <class 'str'>'
```

Note that this error is encountered before ever having to call `string_identity` to reach it. However, it is important to reiterate that the decorator code never inspects the actual function body of `string_identity` here—everything described so far is related to type annotations only.

Another key property of a type system is to ensure all arguments to a function match the type annotated. In this example, the `add_one` function is decorated with the `enforce_type_hints` decorator, and therefore passed as the argument to the decorator. It should be noted that this program does not consider the return type, as a very basic type inference system is required to infer the type of x in the body of the function, but the principles remain the same.

```
1  def enforce_type_hints(func):
2      ann = func.__annotations__
3      ann.pop('return', None)
4  
5      def f(*xs):
6          tps = zip(ann.values(), xs)
7  
8          for (t1,t2) in tps:
9              assert(t1 == type(t2))
10 
11         return func(*xs) #types match, call the original function
12 
13     return f
14 
15 @enforce_type_hints
16 def add_one(x: int) -> int:
17     return x + 1
18 
19 add_one(1)       #OK
20 add_one("foo")   #AssertionError thrown at line 9
```

Running `add_one` with anything but an integer now fails with an `AssertionError`. This happens as the `enforce_type_hints` returns its inner function `f` that receives a list of arguments. In essence what is created here, is a pre-runtime type checker that stops the program if a function annotated with this decorator receives an argument of the wrong type. If it is desired to mimic static checking, the decorator can simply assert the required expressions before returning the function received as an argument:

```
def enforce_int_return_type_hint(func):
    ann = func.__annotations__
    assert(ann['return'] == int)
    return func

@enforce_int_return_type_hint
def add_one(x: int) -> int:
    return x + 1
```

If the type hint of the `add_one` function is not an integer, the program will fail with an assertion error before the body of the function are executed.

2.2.4 AST Inspection

Using Python's built-in `inspect` and `ast` package, it is possible to decompose Python code into its corresponding AST. The AST of the function `add_one` can be seen on listing 2 containing a `Module` with a `FunctionDef` as its body, itself having the name `add_one` accepting x annotated as the `int` type returning a `BinOp`.

Listing 2 AST of add_one

```
Module(
    body=[
        FunctionDef(
            name='add_one',
            args=arguments(
                args=[
                    arg(
                        arg='x',
                        annotation=Name(id='int', ctx=Load()))]),
            body=[
                Return(
                    value=BinOp(
                        left=Name(id='x', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            returns=Name(id='int', ctx=Load()))],
```

Combining this with function decorators it is now possible to do additional inspection of functions before they are run. Python allows this by subclassing the `NodeVisitor` class from the `ast` package that implements the `visitor` design pattern

```python
class Traverser(ast.NodeVisitor):
    def __init__(self, func):
        tree = ast.parse(dedent(inspect.getsource(func)))
        self.visit(tree)

    def  visit_BinOp(self, node: ast.BinOp):
        print(ast.dump(node, indent=4))

Traverser(add_one)
```

Upon visiting a `BinOp` instance, the code in `visit_BinOp` is run, here printing the contents of it. The same applies to other classes in the `ast` package used in AST's such as `Module`, `FunctionDef`, `Return` and so on.

```
BinOp(
    left=Name(id='x', ctx=Load()),
    op=Add(),
    right=Constant(value=1))
```

Checking the AST is a way to do static analysis of Python. In this example one could verify that x is bound to a variable of type `int` and report an error if not.

2.3 Session Types

Session types describe patterns of interactions between either two or more participants in a distributed system in a type-safe way. This allows programmers to specify a communication protocol between participants in distributed systems without worrying about runtime errors related to message passing similar to how types are used to assist programmers when writing programs locally. A *session* is typically associated with a channel or endpoint abstraction that explicitly describes the sequence of actions between participants in a system. The approach in this project deals with a combination of static and dynamic verification, [6] and [8] showed how to do purely dynamic verification of multiparty session types (MPST) in Python. The ability to accommodate both typing disciplines in a meaningful way is among the motivating factors for the typing system introduced in Gradual GV for supporting gradual session types [4].

2.3.1 Binary Session Types

Consider the system on figure 2 modelling a simple system with two participants, A and B. A must first send two integers and then receive an integer. Contrarily, B must first receive two integers and then send an int. These are the local types and describe how each participant should act in the system. Here, following its local type, A first sends the integers 4 and 3, where it then receives 7 from B and stops.

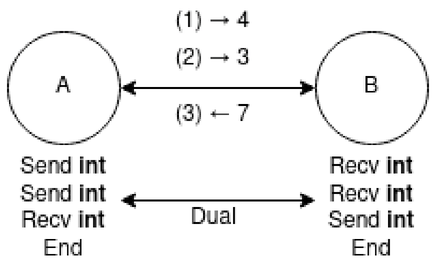


Figure 2: Communication between participant A and B

Following the grammar on listing 3, the formal notation of figure 2 can be seen on listing 4. From left to right: the bang (!) and question mark (?) operators respectively send and receives a value with some type α with its tail being another session type S, a range of labelled choices can be offered with &, and picked among these with $\oplus$, then type variables, recursive types, and lastly the literal for denoting the end of the protocol.

Listing 3 Grammar

$S ::= !\alpha.S \mid ?\alpha.S \mid \&\{l_i : S_i\} \mid \oplus\{l_i : S_i\} \mid \mathbf{t} \mid \mu\mathbf{t}.S \mid \mathbf{end}$

Listing 4 Local types for A and B

A: $!\mathbf{int}.!\mathbf{int}.?\mathbf{int}.\mathbf{end}$
B: $?\mathbf{int}.?\mathbf{int}.!\mathbf{int}.\mathbf{end}$

To prevent a system deadlock, these types must be *dual* meaning one operation should be the inverse of the other, such that when participant A sends something, B must receive something of the same type, this also applies to the offer and choose operators that are dual. If this is not the case, the system may enter a state where both participants are waiting to receive and therefore deadlock.

Another more complicated example is seen on listing 5, a transaction between a diamond seller and a potential buyer. The buyer begins by requesting the transaction, the seller sends a catalogue of colours and prices, and the buyer makes a decision; either by sending the colour, or cancelling the transaction. From the buyer's point-of-view, they will begin by sending a request, receiving a catalogue of items where after they decide to either terminate the transaction (and receive a "goodbye" message), or to purchase a diamond by sending its colour where a receipt is then received. When the seller receives a request, a catalogue is immediately sent by the seller over the channel before waiting for one of two possibilities: either the client will cancel the transaction, or they will send a diamond colour followed by the seller sending a receipt back.

Listing 5 Local types for a *buyer* and *seller*

Buyer: $!\mathbf{str}.?\mathbf{catalogue}.\oplus\{ {}^{\text{decline: } ?\mathbf{str}.\mathbf{end}}_{\text{accept: } !\mathbf{colour}.?\mathbf{str}.\mathbf{end}} \}$

Seller: $?\mathbf{str}.!\mathbf{catalogue}.\&\{ {}^{\text{decline: } !\mathbf{str}.\mathbf{end}}_{\text{accept: } ?\mathbf{colour}.!\mathbf{str}.\mathbf{end}} \}$

2.3.2 Multiparty Session Types

Extending the idea of session types to more than two parties, issues concerning the ordering of types and their dependencies can arise. Consider the example program originally presented in [5]: two buyers B_1 and B_2 that attempt to buy a book from a seller S. B_1 first sends the title of the book to S where after S sends the price to B_1 and B_2. B_1 sends B_2 the amount they are willing to contribute and B_2 then informs S that either: the transaction has been rejected, or B_2 sends S an acknowledgement of the purchase and ends the transaction. The channels between each participant can be written out in formal notation as seen in listing 6, but this does define the causal relationship between the participants.

Listing 6 Local types for the two buyers example

$b_1 b_2$: $!\mathbf{int}.\text{end}$

$b_1 s$: $!\mathbf{string}.?\mathbf{int}.\text{end}$

$b_2 s$: $?\mathbf{int}.\oplus\{ {}^{ok:\ !\mathbf{string}.end}_{ko:\ end} \}$

It is possible for B_1 to tell B_2 its contribution without having first told S what book is wanted which will break linearity. This can be solved by examining the entire system and writing the global protocol down:

$B_1 \to S < str >$.

$S \to (B_1, B_2) < int >$.

$B_1 \to B_2 < int >$.

$B_2 \to S\{ {}^{ok:\ B_2 \to S<str>.end}_{ko:\ end} \}$

And then doing end-point projection (EPP), examining the global protocol, and generating each participant's local protocol as seen on listing 7. Figure 3 provides a better overview

Listing 7 Projections of global type for participants in the two buyers example

B_1: $!S(\mathbf{str}).?S(\mathbf{int}).!B_2(\mathbf{int}).\text{end}$

B_2: $?S(\mathbf{int}).?B_1(\mathbf{int}).!S\oplus\{ {}^{ok:\ !S(\mathbf{str}).end}_{ko:end} \}$

S: $?B_1(\mathbf{str}).B_1!(\mathbf{int}).B_2(!\mathbf{int}).?B_2\&\ \{ {}^{ok:\ ?B_2(\mathbf{str}).end}_{ko:end} \}$

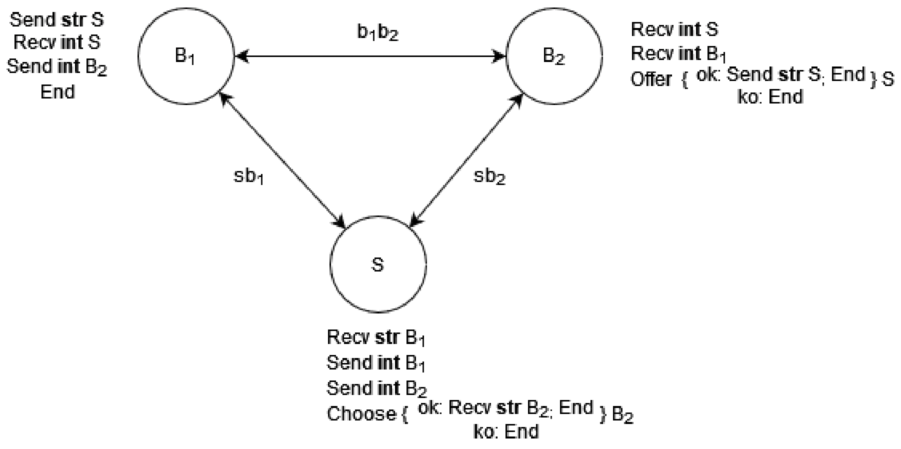


Figure 3: Two buyers

2.3.3 Gradual Session Types

For session types to exist in modern languages that incorporate gradual typing, there should be a way to validate the use of such a session in a gradual way. Session types already exist in a host of static implementations such as Scala, Java, and Go[8] that leverage on strong type systems, to session types in dynamic languages, like Python and Erlang that utilises runtime monitoring.

Session types in gradually typed languages should provide the same flexibility as gradual types, and allow session types to be used in the context of both the static and dynamic type paradigm. The term *gradual session types* as presented in [4] is an implementation that extends the idea of the dynamic type (?) to session types, with a generic session type that abstracts over all session types.

A motivational example from the same paper is to allow two parties to communicate using session types with different type paradigms. The two people, Sy and Rob, are collaborating to build both sides of a *compute service* with the caveat of Sy advocating for static typing, and Rob advocating dynamic typing. The local types of the compute service can be seen below.

Listing 8 Compute service session-types

Client: $\oplus\{$ *neg*: $!\mathbf{int}.?\mathbf{int}.\text{end}$, *add*: $!\mathbf{int}.!\mathbf{int}.?\mathbf{int}.\text{end}$ $\}$
Server: $\&\{$ *neg*: $?\mathbf{int}.!\mathbf{int}.\text{end}$, *add*: $?\mathbf{int}.?\mathbf{int}.!\mathbf{int}.\text{end}$ $\}$

The server specified by the session type offers a choice between two arithmetic operations: addition and negation. It either reads one or two integers depending on the operation, and sends back a result. The client must choose one of these operations with a label (one of {'add', 'neg'}) and send the corresponding number of integers.

[8]https://groups.inf.ed.ac.uk/abcd/session-implementations.html

Sy implements the client in the statically-typed language, GV, with the type of ComputeDneg defined as $\oplus$neg : **!int**.?**int**.end?. The implementation can be seen below.

```
negationClient : int → ComputeDneg → int
negationClient v c =
    let c = select neg c in
    let c = send v c in
    let y,c = receive c in
    let _ = wait c in
    y
```

The `negationClient` accepts an integer (v) and a channel where it first selects the 'neg' label, and proceeds to send an integer, receive the negated integer, and then close the connection adhering to the session type. On the other hand, Rob implements the server in dynamically-typed Unityped GV:

```
dynServer c =
    case c of {
        neg: c . serveOp 1 ( \x. −x) c;
        add: c . serveOp 2 ( \x y. x+y ) c
    }

serveOp n op c =
    if n == 0 then
        close (send op c)
    else
        let v, c = receive c in
        serveOp (n−1) (op v) c
```

The main `dynServer` function accepts a channel and waits for an incoming label. Depending on this label, the auxiliary `serveOp` function is called that accepts an integer, a function and a channel on where to receive and send. The `serveOp` function will then count down n and for each iteration receive on the channel until the parameter n is zero and a final send is performed and the channel is then closed. A key thing to note is that `serveOp` might receive session types with varying signatures which is allowed dynamically, but is problematic from a statically typed point-of-view.

It is not immediately clear how this communication should be gradually checked. Their suggestion includes the use of casts and blame labels, and will be discussed in section 4.2.

2.4 Language Engineering

Lexing (or scanning) is the process of reading characters (lexemes) from a text file and assembling them into a stream of tokens which is then passed along to a parser. The parser itself is a program that builds an abstract syntax tree (AST) according to a specified grammar from a stream of tokens. Static checks can hereafter be performed on the AST and evaluation or compilation semantics can be defined. A typical flow for an interpreted language can be seen on figure 4.
Considering a subset of the structured query language (SQL) and its table definition that allows arbitrarily many columns in a table:

```
CREATE TABLE tab {
    Column1,
    Column2,
    ...
    ColumnN
}
```

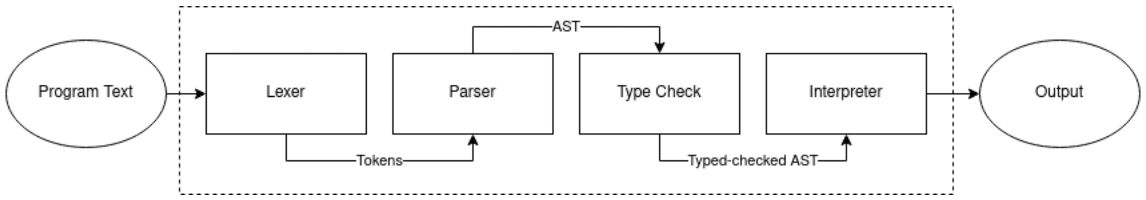


Figure 4: Program flow

The lexer will generate a sequence of lexemes (tokens):

```
CREATE, TABLE, IDENTIFIER tab, LEFT_BRACE, SEMICOLON, IDENTIFIER Column1, COMMA,
IDENTIFIER Column2, COMMA, ... , RIGHT_BRACE
```

That can be assembled into an AST by the parser. Firstly, the grammar of a language defines the valid sentences allowed. This is typically done by specifying a set of rules on how tokens can be arranged to form valid statements. An example of this is the Extended Backus-Naur Form (EBNF) which is composed of production rules, terminals, and rule calls. Its grammar can then take the form:

```
Table = "CREATE" "TABLE" ID "{" Column ("," Column)* "}";
Column = ID;
ID = [A-Z|a-z|_][A-Z|a-z|0-9|_]*;
```

This grammar states that to define a table, it must begin with the string literals "CREATE TABLE" followed by a rule call to the `identifier` rule which simply accepts strings starting with the alphabetic letters (both upper and lowercase), or an underscore. Hereafter a pair of curly braces should be matched which should contain one or more columns. This is done by again making a rule call to the `Column` rule. This recursive definition allows arbitrarily many columns inside a table definition and is used in combination with the lexemes generated by the lexer, to construct the AST. The AST can then be validated by a typechecker that either rejects or accepts it, passing it along to the interpreter if successful. The interpreter can then finally directly execute the AST and produce an output.

2.5 Scribble

Scribble is a protocol description language that, for some system, represents an agreement by two or more parties on how they should communicate with each other in a predefined order [9]. The goal of a Scribble protocol is to generate session types that can be used in general-purpose programming languages. A Scribble protocol is firstly defined in terms of a global protocol. This global protocol provides a birds-eye view of the scenario and describes how all roles interact with each other, in the form of a series of message exchanges where the low-level identity of the roles has been abstracted away. Scribble enables the description of protocols using message passing, choice, and recursion as can be seen on listing 9 that models the previous two buyer's example.

Listing 9 Global protocol for the two buyers example

```
global protocol TwoBuyers(role B1, role B2, role Seller) {
    Title(Message) from B1 to Seller;
    Quote(Price) from Seller to B1;
    Quote(Price) from Seller to B2;
    Amount(Price) from B1 to B2;

    choice at B2 {
        Address(Message) from B2 to Seller;
    } or {
        Quit();
    }
}
```

The first line declares a protocol by the name of `TwoBuyers` having three participating roles. This protocol consists of a series of message exchanges between the participants before a choice has to be made by B_2 that *Seller* should act on. Once the protocol has been defined, it can be checked using Scribble tools already developed[9] to ensure that it is well-formed and deadlock free. This verification is necessary as not all protocols are safe and well-typed. Figure 5 describes the Scribble workflow.

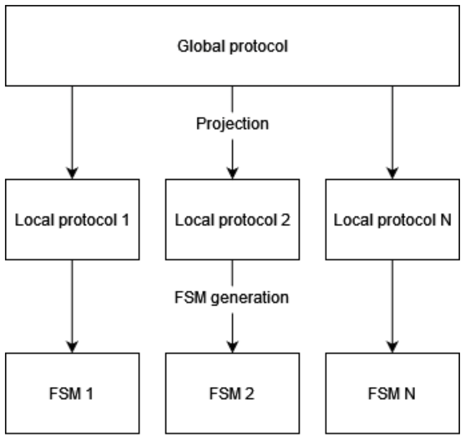


Figure 5: Scribble workflow

A global Scribble protocol is projected into a local protocol for each role defined. This generation of local roles corresponds to the MPST local types that describe the protocol from that particular roles' point of view. The global protocol on listing 9 is projected into a local protocol for each role as seen on listing 10 where the local protocol for the *Seller* is shown. These local protocols describe how the individual participant interacts with the other participants. Lastly, the local protocol is converted into a finite state machine.

[9]`httpfs://github.com/scribble/scribble-java`

Listing 10 Local protocols

```
local protocol TwoBuyers at Seller(role B1, role B2, role Seller) {
    Title(Message) from B1;
    Quote(Price) to B1;
    Quote(Price) to B2;

    choice at B2 {
        Address(Message) from B2;
        end
    } or {
        end
    }
}
```

FSM generation Similar to the local protocol, a finite state machine (FSM) in this context is another representation of the individual roles' point of view. This abstraction from protocol to state machine comes naturally due to firstly, the constraint on session types that can only be in one state at one time, the possibility of repeating protocols, and lastly branching.

Grammar The Scribble grammar can be seen on listing 11. It specifies the grammar for both global and local protocols. Firstly, the global protocol is defined by the rule P which states that global protocols must start with the string literals "global protocol" followed by any identifier. Hereafter the roles of the protocol are listed inside a pair of parentheses where each role must be prefixed with "role". The body of the protocol is a call to the G rule inside a pair of curly parentheses. The G rule describes the valid statements of the global protocols, starting off with the most common operation that sends `S` with a label a, from one role to another. This is then followed by another statement. The second statement is the choice that must be made in some role A. The choice can contain two or more branches of statements defined by a pair of curly parentheses with a recursive call to G. Next, recursion is defined using the `rec` keyword followed by a label t. This is then followed by a pair of curly braces that contains more statements. Lastly, to jump to a label specified by `rec`, the keyword `continue` must be used with a specific label.

The local protocol similarly has a rule L that defines a local protocol from the perspective of some role A_i. This is again followed by the roles in the protocol and a series of statements, this time using the T rule inside a pair of curly parentheses. Contrary to G, T specifies a few more statements. Firstly, sending and receiving a message is unidirectional and only specifies the recipient as the sender is always implied to be the current role. The choice, recursion and continue are identical to those of the global protocols except for the statements they contain. Lastly, local protocols are stopped with the `end` keyword.

Listing 11 Scribble grammar

$\langle P \rangle$ ::= global protocol *pro* (role A_1, ..., role A_N) { $\langle G \rangle$ }

$\langle G \rangle$ ::= a(S) from A to B; $\langle G \rangle$
| choice at A { $\langle G \rangle$} or ... or { $\langle G \rangle$ }
| rec t { $\langle G \rangle$ }
| continue t

$\langle L \rangle$ ::= local protocol *pro* at A_i (role A_1, ..., role A_N) { $\langle T \rangle$ }

$\langle T \rangle$ ::= a(S) to B; $\langle T \rangle$
| a(S) from B; $\langle T \rangle$
| choice at A { $\langle T \rangle$} or ... or { $\langle T \rangle$ }
| rec *pro* { $\langle T \rangle$}
| continue *pro*
| end

3 SessionPy

This section presents the type system and implementation of SessionPy. In section 3.1 the formal rules of the type system are presented. In section 3.2 the implementation of the type checker, endpoint, session types and, lastly, how Scribble protocols are projected into Python files, are presented.

To keep the project manageable in scope, a subset of Python has been the focus of SessionPy. The subset is Turing-complete and thus provides enough logic constructs to allow for session types to be used in meaningful ways. This subset is described by the grammar in listing 12. The grammar is made out of five rules: P for program (or statement), e for expressions, t denotes types, and the last two are for Python's `match` statement.

The first form P can take, is the separation of two statements with a semicolon. While this is not entirely representative of Python as it does not use semicolons it can be thought of as a newline. This simply allows chaining multiple statements together to form a program. The next is an if-then-else statement, that has an expression as the guard and two statements that may be run depending on e. Variable assignment is also possible, assigning an expression to an identifier that potentially can have a type hint. Loops are also supported in the form of `for` and `while` that repeatedly executes a statement. Function declaration is also allowed having a name, a list of parameters, and a body. Using the communication primitive, the endpoint it is possible to use the `send` and `choose` operation on it, each accepting an expression. The Python `match` statement is supported in two forms: the normal one, matching on the result on some expression against several cases, and a specialised case where the subject of the match statement is the `offer` operation of an endpoint. The next two are function, and method application invoking a *callable* with some arguments. Lastly, a statement may also be a simple expression that can be a `recv` operation, binop, unop, a lambda, a variable, or a constant. The *typ* rule represents the supported types which are any basic type (str, int, bool), tuples, and parametric types such as lists or dictionaries.

Listing 12 Chosen Python subset

$\langle P,\ Q\rangle$ ::= $\langle P\rangle$;$\langle Q\rangle$
| `if` e `then` $\langle P\rangle$ `else` $\langle Q\rangle$
| var = $\langle e\rangle$
| var: $\langle t\rangle$ = $\langle e\rangle$
| `for` var `in` $\langle e\rangle$: $\langle P\rangle$
| `while` $\langle e\rangle$: $\langle P\rangle$
| `def` $var(parameters)$: $\langle P\rangle$
| $endpoint$.`send`($\langle e\rangle$)
| $endpoint$.`choose`($\langle e\rangle$)
| `match` $\langle e\rangle$: $\langle cases\rangle$
| `match` $endpoint$.`offer`(): $\langle cases\rangle$
| $var(arguments)$
| $var.var(arguments)$
| $\langle e\rangle$

$\langle e\rangle$ ::= $endpoint$.`recv`()
| $\langle e\rangle$ $binop$ $\langle e\rangle$
| $unop$ $\langle e\rangle$
| `lambda` $parameters$: $\langle e\rangle$
| var
| $constant$

$\langle t\rangle$ ::= $basic\ type$ | `List`[$\langle t\rangle$] | `Dict`[$\langle t\rangle$, $\langle t\rangle$] | ($\langle t\rangle$, $\langle t\rangle$)

$\langle cases\rangle$ ::= $\langle case\rangle$+

$\langle case\rangle$ ::= `case` var: $\langle P\rangle$

3.1 Type System

To define the type rules for the type checker, two environments are used; Γ (gamma) that maps variables, including primitives and function definitions, to types, and Σ (sigma) that is the linear environment wherein session types are mapped to its state. Since it's crucial *how* the session types are manipulated, the progress like $\Sigma \rhd P \rhd \Sigma'$, where the program P *may* have altered the session types contained is read as "linear environment before and after executing P". Particularly, this specification requires that a sigma with no ticks before executing P, and zero ticks after execution should be in the same state, however, where environments with different numbers of ticks are allowed to vary, they *may* still be in identical states.

The semi-colon rule seen in equation 2 firstly evaluates P in some linear environment, advancing it to Σ''. Hereafter Q is evaluated in Σ'', ending up with Σ'

$$\frac{\Gamma \vdash \Sigma \rhd P \rhd \Sigma'' \quad \Gamma \vdash \Sigma'' \rhd Q \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd P; Q \rhd \Sigma'} \tag{2}$$

The if-then-else statement in equation 3 shows that the guard must have a boolean type, whereas the *then* and *else* branches, specifically the procedures P and Q, should affect the linear environment uniformly. If not, a violation of some session type has occured.

$$\frac{\Gamma \vdash \Sigma \rhd e : \mathbf{bool} \rhd \Sigma'' \quad \Gamma \vdash \Sigma'' \rhd P \rhd \Sigma' \quad \Gamma \vdash \Sigma'' \rhd Q \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd \text{if } e \text{ then } P \text{ else } Q \rhd \Sigma'} \tag{3}$$

The assign statement in equation 4 evaluates the expression and binds the target to the type of the resulting expression. In general, evaluation of expressions, like e, may affect session types, and therefore Σ' are to be expected after this evaluation.

$$\frac{\Gamma \vdash \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd x = e \rhd \Gamma[x \mapsto \boldsymbol{\alpha}], \Sigma'} \tag{4}$$

Annotated assignments are similar as seen in equation 5, but the binding from x to the annotated type are used instead of the resolved type.

$$\frac{\Gamma \vdash \Sigma \rhd e \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd x : \boldsymbol{\alpha} = e \rhd \Gamma[x \mapsto \boldsymbol{\alpha}], \Sigma'} \tag{5}$$

The *for* loop as seen in equation 6 iterates over some collection. At each iteration, after executing P the ending state should be equal to the the starting state.

$$\frac{\Gamma \vdash x : \boldsymbol{\alpha} \quad \Gamma \vdash \Sigma \rhd xs : \boldsymbol{\alpha} \text{ list} \rhd \Sigma' \quad \Gamma \vdash \Sigma' \rhd P \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd \text{for } x \text{ in } xs \text{ do } P \rhd \Gamma[x \mapsto \boldsymbol{\alpha}], \Sigma'} \tag{6}$$

The *while* loop seen in equation 7 behaves exactly as the *for* loop without any binding and the iterable expression.

$$\frac{\Gamma \vdash \Sigma \rhd e : \textbf{bool} \rhd \Sigma' \quad \Gamma \vdash \Sigma' \rhd P \rhd \Sigma}{\Gamma \vdash \Sigma \rhd \text{while } e \text{ do } P \rhd \Gamma', \Sigma} \tag{7}$$

Function definitions are added to the current environment with the types annotated as seen in equation 8. A function definition may have arbitrarily many parameters of different types, this is denoted $(p_1, ..., p_n)$. The current environment is extended with a binding from the function name to its type.

$$\frac{f : \textbf{str} \quad p_1 : \boldsymbol{\alpha_1}, ..., p_n : \boldsymbol{\alpha_n} \quad r : \boldsymbol{\beta} \quad \Gamma[p_1 \mapsto \boldsymbol{\alpha_1}, ..., p_n \mapsto \boldsymbol{\alpha_n}] \vdash \Sigma \rhd P \rhd \Sigma'}{\Gamma \vdash \text{def } f(p_1, ..., p_n) \rightarrow r \text{ do } P \rhd \Gamma[f \mapsto \boldsymbol{\alpha_1} \rightarrow ... \rightarrow \boldsymbol{\alpha_n} \rightarrow \boldsymbol{\beta}]} \tag{8}$$

The first session type operation *send* as seen in equation 9 states that it must be in a state where a send operation is expected denoted by $\Sigma \xrightarrow{!\boldsymbol{\alpha}} \Sigma'$, and that the type of the payload has type $\boldsymbol{\alpha}$.

$$\frac{\Gamma \vdash \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'' \quad \Sigma'' \xrightarrow{!\boldsymbol{\alpha}} \Sigma'}{\Gamma \vdash \Sigma \rhd \text{send } e \rhd \Sigma'} \tag{9}$$

The *choose* operation (equation 10) is similar to *send* but the expression must be of *str* type (label) and the session type should be in a state where the next operation is *choose*.

$$\frac{\Gamma \vdash e : \textbf{str} \quad \Sigma \xrightarrow{\&\textbf{str}} \Sigma'}{\Gamma \vdash \Sigma \rhd \text{choose } e \rhd \Sigma'} \tag{10}$$

The *match* statement (equation 11) is similar to the if-then-else statement; the state after evaluation of any branch should be identical.

$$\frac{\Gamma \vdash \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'' \quad \Sigma'' \rhd P_1 \rhd \Sigma' \quad \Gamma \vdash \Sigma'' \rhd P_n \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd \text{match } e \text{ case } c_1 \; P_1, \; ..., \; \text{case } c_n \; P_n \; \rhd \Sigma'} \tag{11}$$

The *offer* operation (equation 12) expects the session type to be in a state where the next operation is *offer* with options T_1 through T_n. The expression results in a *str* type and should only be used as the subject of a match statement.

$$\frac{\Sigma \xrightarrow{\oplus[T_1, ..., T_n]} \Sigma'}{\Gamma \vdash \Sigma \rhd \text{offer}[T_1, ..., T_n] : str \rhd \Sigma'} \tag{12}$$

Function call performs a lookup of the called function and ensures that the type of the arguments match the parameters. The result of the expression is the return type of the function. This rule is in curried form meaning that it could have any number of arguments for a function of arbitrary arity.

$$\frac{\Gamma(f) = \boldsymbol{\alpha} \rightarrow \boldsymbol{\beta} \vdash \Sigma \quad \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd f\ e : \boldsymbol{\beta} \rhd \Sigma'} \tag{13}$$

Method calls behave much like function calls. It performs a lookup on the class where the method must be included in the set of members. The arguments must match the parameters, and the return type of the entire expression is the return type of the method.

$$\frac{f \text{ member of } x \quad \Gamma(f) = \boldsymbol{\alpha} \rightarrow \boldsymbol{\beta} \vdash \Sigma \quad \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd x.f\ e : \boldsymbol{\beta} \rhd \Sigma'} \tag{14}$$

The *receive* operation (equation 15) must be in a state where it is expected to receive some $\boldsymbol{\alpha}$ and also results in a type $\boldsymbol{\alpha}$

$$\frac{\Sigma \xrightarrow{?\boldsymbol{\alpha}} \Sigma'}{\Gamma \vdash \Sigma \rhd \text{recv} : \boldsymbol{\alpha} \rhd \Sigma'} \tag{15}$$

The unop (equation 16) returns its own type.

$$\frac{op \in \{\text{not}, -, +, \sim\} \quad \Gamma \vdash \Sigma \rhd e : \boldsymbol{\alpha} \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd op\ e : \boldsymbol{\alpha} \rhd \Sigma'} \tag{16}$$

Binary operations (equation 17) unifies the types of both input types, and returns the most specialised type possible.

$$\frac{op \in \{+, -, *, /\} \quad \Gamma \vdash \Sigma \rhd e_1 : \boldsymbol{\alpha} \rhd \Sigma'' \quad \Gamma \vdash \Sigma'' \rhd e_2 : \boldsymbol{\alpha} \rhd \Sigma'}{\Gamma \vdash \Sigma \rhd e_1\ op\ e_2 : \boldsymbol{\alpha} \rhd \Sigma'} \tag{17}$$

The lambda (equation 18) is an expression with a function return type specified by its parameters. Similar to function definition, a lambda can have arbitrarily many parameters.

$$\frac{\Gamma[p_1 : \boldsymbol{\alpha_1}, ..., p_n : \boldsymbol{\alpha_n}] \vdash \Sigma \rhd e : \boldsymbol{\beta} \rhd \Sigma'}{\Gamma \vdash \text{lambda } p_1, ..., p_n \text{ do } e : \boldsymbol{\alpha_1} \rightarrow ... \rightarrow \boldsymbol{\alpha_n} \rightarrow \boldsymbol{\beta}} \tag{18}$$

Looking up a variable x, it is found in the current environment and the corresponding type is returned (equation 19).

$$\frac{\Gamma(x) = \boldsymbol{\alpha}}{\Gamma \vdash x : \boldsymbol{\alpha}} \tag{19}$$

The constant is a literal value representing integers, strings, booleans and the None value (equation 20)

$$\frac{}{c : \boldsymbol{\alpha}} \tag{20}$$

3.2 Python Implementation

This section will present the technical implementation of SessionPy. First, introducing typing constructs used throughout the implementation followed by examining the process from syntactic definition of session types to finite state machines (FSM). These state machines are used in combination with the type checker to validate session types. Lastly, everything is put together and the projection from protocol to Python is shown.

3.2.1 Typing Constructs and Unification

Streamlining the notion of a *type* for the type checker, a combination of various typing constructs under one umbrella term `Typ` is used throughout the implementation as defined in listing 13. This is done to address the nature of classes, types, and objects discussed in section 2.2 since they all share the commonality of *being* a type, but whereas in Python they are separate constructs (recall from 2.2 that `list[int]` is a `GenericAlias` and `int` is a `type`, but they are both *types*). Functions share their type with a common list that allows for an arity of arbitrary length where the last element denotes the return type. For example, the signature of a function accepting two integers and returning a string will be represented as the list `[int, int, str]`. As described in section 2.2.2, parameterised generics correspond to various generic alias types, and in the case of the type system these constructs are unified under one term. For technical reasons, class types are synonymous with strings. Finally, a `Typ` is simply the union of these constructs including the base type for all built-in classes `type` making functions and basic types along with parameterised generics belong to the same family.

Listing 13 Typing definition used in SessionPy

```
FunctionTyp = list
ParameterisedType = Union[typing._GenericAlias, GenericAlias,
 ↪  typing._SpecialGenericAlias]
ClassTypes = str
Typ = Union[type, FunctionTyp, ParameterisedType, ClassTypes]
```

To accommodate subclassing and implicit conversion of types as discussed in section 2.2, unification of two or more types are also allowed; the *reduction* of multiple types into a single type of the most generic kind. If $B <: A$ (B is a subclass of A) then the unification should result in A; one such built-in relation is `int` <: `float` where the union is a `float`. This idea is used in the type checker described in section 2.4 for validation of the various constructs from binary operations to annotated lists to make sure types are consistent.

3.2.2 Session Type Representation

The concrete syntax used for writing out session types in Python is an important aspect of the project that enables us to stay very close to the semantics of how MPST are written using formal notation. This is possible due to recent language extensions presented in 2.2.2. The session types are represented as parameterised generics that allow for a recursive type variable for denoting "the rest of the session type". Furthermore, they can contain other types (for sending or receiving data), dictionaries for branching, and should include an actor that is simply a string expected by the generic type. The grammar for these types are seen in listing 14.

Listing 14 Grammar for concrete session type syntax in SessionPy

$$
\begin{aligned}
\langle st \rangle \quad ::= \;& \texttt{Send[}\textit{type}\texttt{, }\textit{actor}\texttt{, }\langle st \rangle\texttt{]} \\
\mid\;& \texttt{Recv[}\textit{type}\texttt{, }\textit{actor}\texttt{, }\langle st \rangle\texttt{]} \\
\mid\;& \texttt{Label[}\textit{label}\texttt{, }\langle st \rangle\texttt{]} \\
\mid\;& \texttt{Offer[}\textit{actor}\texttt{, \{ }\langle \textit{mapping} \rangle + \texttt{ \}]} \\
\mid\;& \texttt{Choose[}\textit{actor}\texttt{, \{ }\langle \textit{mapping} \rangle + \texttt{ \}]} \\
\mid\;& \texttt{End}
\end{aligned}
$$

$$
\langle \textit{mapping} \rangle ::= \textit{label}\texttt{: } \langle st \rangle\texttt{,}
$$

Listing 15 Definition of type variables including the Send and Choose session type class

```
A = TypeVar('A')
ST = TypeVar('ST')
Actor = TypeVar('Actor', str)

class SessionType: ...

class Send(SessionType, Generic[A, Actor, ST]): ...

class Choose(SessionType, Generic[Actor, A]): ...
```

Listing 15 shows the interplay between type variables, parameterised generics, and subtyping that makes up the syntax of session types. All session types belong to the same family by inheriting from the `SessionType` class. The `Generic` superclass inherited by all session types is what determines the number of generic parameters to the type. Since identical type variables to this generic are prohibited, `A` denoting *any* (generic) type and `ST` denoting, recursively, another session type are different. The branching constructs expect a dictionary as their second type parameter which is not immediately clear from listing 15, but this check will not be made when validating the session type as part of FSM generation. The session type written in formal notation: $!\text{Alice}(\textbf{int}).\text{Bob}\oplus\{\begin{smallmatrix} A\text{:} & !\text{Bob}(\textbf{int}).\textbf{end} \\ B\text{:} & ?\text{Bob}(\textbf{str}).\textbf{end}\end{smallmatrix}\}$ can then be expressed in Python:

```
Send[int, 'Alice', Choose['Bob',
    { 'A': Send[int, 'Bob', End],
      'B': Recv[str, 'Bob', End }]]]
```

Labels provide a way of defining recursive session types. That is, if looping behaviour is required, labels can be used to name a specific state to jump back to. Consider the following session type first written in formal notation, then as a type in SessionPy.

$$?\text{Alice}(\textbf{bool}).\mu\textbf{t}.?\text{Alice}(\textbf{int}).?\text{Alice}(\textbf{bool}).\textbf{t}$$

```
Recv[bool, 'Alice', Label['loop', Recv[int, 'Alice', Recv[bool, 'Alice', 'loop']]]]
```

The 'loop' label is used to mark the fixpoint after a bool has been received from Alice, and the 'loop' at the end of the session type jumps back to 'loop'.

3.2.3 State Machine

Structurally, a session type resembles a finite automaton in that it always exists at one point (state), and can make deterministic progress through the type from input. For this reason, SessionPy uses state machines to model types; a directed graph where the vertices represent the current *state* of the automata, and the edges represent the legal transitions that can be taken from this state. For a session type to be statically valid and sound, it is required that its usage precisely follows the series of edges generated from the type.

The state machine is generated from the syntactic type as presented in the previous section. The `State` class can be thought of as the current entry point in the FSM that has a unique identifier, a flag whether it is an accepting state (or *any* state, explained in a while), and the outgoing transitions from it—all shown in listing 16.
Each transition in the FSM is aptly represented by the `Transition` class—the keys in the `outgoing` dictionary seen in listing 16—and its definition can be found in listing 17.

Listing 16 State class constructor and attributes

```
class State:
    def __init__(self, identifier: int, accepting_state: bool = False) -> None:
        self.identifier: int                    = identifier
        self.accepting: bool                    = accepting_state
        self.any_state: bool                    = False
        self.outgoing: dict[Transition, State] = {}
```

Listing 17 Transition class constructor and attributes

```
class Transition:
    def __init__(self, action: Action) -> None:
        self.action: Action = action
        self.actor: str | None = None
        if action in [Action.SEND, Action.RECV]:
            self.typ: Typ = Any
        elif action == Action.BRANCH:
            self.branch_options: dict[str, State] = {}
        elif action == Action.LABEL:
            self.name = ''
            self.st = None
```

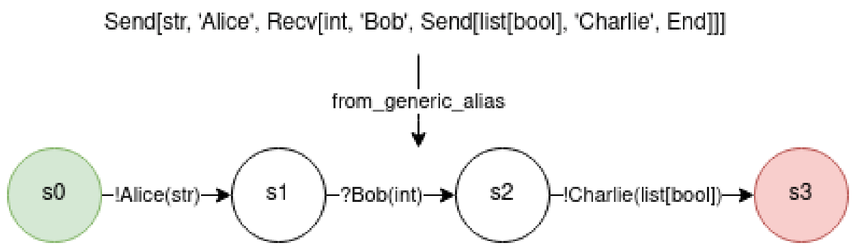
The `Transition` class exploits the dynamic nature of Python; depending on what `Action` argument is passed to the constructor, different attributes are set for that object. Common is that all transitions need an action—one of the options seen in listing 18—and an optional name on the actor that is the other entity communicated with. For sending and receiving data, additionally a `Typ` is required. If branching out into multiple states, it is required to maintain a dictionary of the option picked with a mapping from a string (the label) to State (the session type).

Listing 18 Action enums as part of transitions

```
class Action(str, Enum):
    SEND = 'send'
    RECV = 'recv'
    LABEL = 'label'
    BRANCH = 'branch'
```

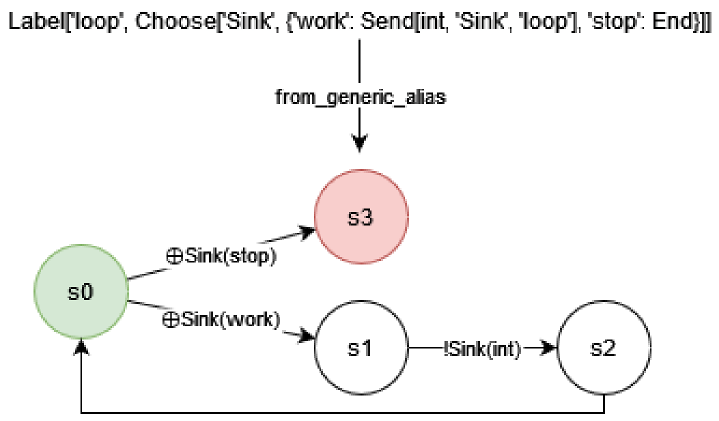
Figure 6 visualises how, from the syntactic definition of a session type, written out as a parameterised type, to a FSM represented by the `State` class is generated. The red colour indicates an *accepting* state.

Figure 6: Visualisation of FSM generation from syntactic definition


Send[str, 'Alice', Recv[int, 'Bob', Send[list[bool], 'Charlie', End]]]
from_generic_alias
s0
!Alice(str)
s1
?Bob(int)
s2
!Charlie(list[bool])
s3

Loops and branching defined using the syntax of SessionPy can also be turned into a state machine as seen in figure 7. The edge back to `s0` from `s2` showcases the jump back to the repeating state. Two edges out of the `s0` represents the choice.

Figure 7: Visualisation of FSM with branching and recursion generated from syntactic definition


Label['loop', Choose['Sink', {'work': Send[int, 'Sink', 'loop'], 'stop': End}]]
from_generic_alias
s3
⊕Sink(stop)
s0
⊕Sink(work)
s1
!Sink(int)
s2

3.2.4 Typechecker

In order to validate a session type, a static inspection of the code is required to ensure type usage is consistent with the respective endpoints. For this reason, SessionPy implements a gradual type checker. Utilising the visitor pattern from `NodeVisitor`, the type checker has been implemented by visiting all of the AST nodes part of the Python subset shown in listing 12 including additional nodes such as break, return and compound assignments. The full list of visited AST nodes and a brief explanation can be seen in table 1 in the appendix.

Given that the list of AST nodes ranges from statements to expressions and miscellaneous Python-specific constructs, different return types (if any) are expected from a visit to them. For instance, a visit to `ast.Compare` node should validate that the operands are unifiable and comparable, and lastly return a boolean. Visiting a looping construct like `ast.For` should visit its test condition, the expression it ranges over (typically, a list) and the body of statements, but it has no meaningful return value like comparison. To show the interconnection between the typing rules presented in 3.1 and the concrete implementation of the type checker, the validation process of loops (*while* and *for*) are presented. The example found in listing 19 demonstrates the visitor pattern in action; statements contained in the loop body, and the test for *while* loops, are recursively visited on lines 9 and 10. These calls can make a visit to *any* AST nodes from the subset which takes care of nested constructs and add any new bindings that they may contain. The environment corresponding to Σ in the typing rules is the one obtained by extracting all `State` objects from the main environment Γ. In listing 19, this happens concretely on lines 2 and 11. Recall that loops are validated and sound if the linear environment is the same after looping—that is, any state machine entering a loop should be in this exact state immediately after exiting. This validation is done as the last step, expecting that its identifier can be found in the set of entrypoints.

Listing 19 Validation of loops in the typechecker

```
1  def validate_loop(self, node: While | For):
2      pre_endpoints = self.env().get_kind(State) # sigma
3      if pre_endpoints:
4          eps = [ep[1] for ep in pre_endpoints]
5          for current_state in eps:
6              self.env().loop_entrypoints.add(current_state.identifier)
7          pre_endpoints = eps
8      if isinstance(node, While):
9          self.visit(node.test)
10     self.visit_statements(node.body)
11     post_endpoints = self.env().get_kind(State) # sigma
12     if pre_endpoints and post_endpoints: # post-validation
13         post_endpoints = [eps[1] for eps in post_endpoints]
14         for post_ep in post_endpoints:
15             ep_id = post_ep.identifier
16             expect(ep_id in self.env().loop_entrypoints or ep_id in
               ↪  self.env().loop_breakpoints,
17                 f'loop error: id #{ep_id} not in entry or breakpoints')
```

Static validation of endpoints After the type checker has run, all endpoints defined in the type checked source file are expected to be in an *accepting* state, implying that it has progressed to the end (the type is *exhausted*). Consider the example presented in listing 20.

Listing 20 Example program where one endpoint (ep2) is not in an accepting state

```python
ep1 = Endpoint(Send[str, 'Alice', Recv[int, 'Bob', End]], routing_table)
ep2 = Endpoint(Recv[bool, 'Charlie', Send[list[str], 'Mallory', End]],
↪  routing_table)

ep1.send("hello")
cond = ep2.recv()
if cond:
    i = ep1.recv()
else:
    i = ep1.recv() + 1

"""
raises SessionException:
endpoint <ep2> not exhausted - next up is:
- send list[str] @ Mallory
"""
```

In this example, endpoint `ep1` is done (or *exhausted*) whereas `ep2` still needs to send a list of strings to Mallory which the static checker therefore rejects before the program is run, telling us what next action should be taken.

Progress is made through the state machine upon visits to the Attribute node during AST inspection. From table 1 it was shown that attribute access is what happens when dot notation is used on some object. Specifically, during AST inspection, the form `<endpoint>.<call>` is checked for. If this attribute is present, its `call` attribute is matched on, according to the session type API in Endpoint: `send(expr)`, `recv()`, `choose(label)`, `offer()`, and do validation and progress accordingly. To underline exactly how this happens, the implementation and validation of the `<endpoint>.send(expr)` is highlighted in listing 21.

Listing 21 Implementation of send during static analysis

```python
argument = args.head()
valid_action, valid_typ = nd.valid_action_type(op, argument)
expect(valid_action, f'expected a {nd.outgoing_action()}, but send was called',
↪  node)
expect(valid_typ,
       f'expected to send a {type_to_str(nd.outgoing_type())}, got
        ↪  {type_to_str(argument)}',
       node)
next_nd = nd.next_nd()
self.env().bind_var(ch_name, next_nd)
```

In this example, the `argument` will be a concrete type evaluated from the `expr` passed to `send(expr)`. Alongside helper functions located in the `State` class, this checks if the type and operation is what is expected from the current point in the FSM. If both the action and type are valid, progress is made to the next state, and rebind the endpoint to this updated state.

Stub notation for functions — the Any state Passing endpoint objects to functions is possible by either annotating the parameter with the full session type, or by using stub notation denoted by `...` (called *ellipsis*). This construct is typically used interchangeably with the keyword `pass` to denote the inaction; often used as placeholder for branches or functions that need to be implemented. In this case, it can be denoted "the rest of the session type". The session type `Recv[bool, 'Bob', Send[list[str], 'Alice', End]]` is similar, or in fact equal, to `Recv[bool, 'Bob', ...]`. Expectedly, the cost of this notation is the lack of information provided statically; in the latter session type no information on what happens after a boolean is received from Bob is present, but statically it is allowed for *anything* to happen. To exemplify this see listing 22 where a function uses this notation.

Listing 22 Example program demonstrating stub notation for types

```python
ep = Endpoint(Send[str, 'Alice', Recv[int, 'Bob', Send[list[str], 'Alice', End]]],
↪  routing_table)

def f(c: Recv[Any, 'Bob', ...]) -> Any:
    return c.recv()

ep.send("hello")
i = f(ep)
ep.send(["goodbye", "for", "now"])
```

The example is statically sound since the function `f` expects an endpoint that next should receive a value of type `Any`—the supertype of all classes—and that it should receive this value from Bob, which it does.

This works by comparing the passed endpoints current (remaining) type to that specified by the function's parameters, and if the ellipsis is reached during this validation, it will accept the passed endpoint's type no matter how the tail differs. What is of importance, as again can be seen in listing 22 is that the session type prefixes are identical up until the ellipsis.

3.2.5 Endpoints

To support doing anything useful with session types, an underlying implementation responsible for handling the communication must be implemented. More specifically it can be used in inter-process communication and over networks. Likewise, sending and receiving should block the executing thread until the other party is ready to perform the dual operation.

The implementation of the `Endpoint`, is built on Pythons low-level networking interface; the socket, and exposes a very simple API: `Send`, `Recv`, `Choose` and `Offer`. Given a session type and a routing table, an endpoint can be instantiated and the local protocol for the current role is ready to be implemented. The routing table is a Python dictionary that maps each role of the protocol to their respective address to handle the communication. An example can be seen below.

```python
routing_table = {'self': ('localhost', 5000), 'B2': ('localhost', 5001), 'Seller':
↪  ('localhost', 5002),}

ep = Endpoint(Send[str, 'Seller', Recv[float, 'Seller', Send[float, 'B2', End]]],
↪  routing_table)
```

Once an endpoint has been instantiated, a state machine is built from the provided session type to support runtime verification as the endpoint is used throughout the program.

Dynamic verification is performed by the state machine object that was generated as part of the static analysis. This is stored in each `Endpoint` object, and used to verify the communication at runtime. This monitoring works similarly to the actions taken during the static analysis by progressing in the state machine, but this happens without any knowledge about the source program; each endpoint simply validates that the current API call adheres to the session type, and that the data types are consistent with it too.

Send sends a message to the role of the next operation, defined by the session type. In the above example, that would be the 'Seller' role as the first action required by the session type is to send a string to the 'Seller' role. To start, the state machine must be in a state where a send is expected, this is done by looking at the generated state machine and from the current state, finding the outgoing transition. If this transition is not valid an error is raised, otherwise, the receiving role's address is found in the routing table, the message is sent and the state machine progresses. This is seen on the code snippet below, where the next role `actor` is firstly found in the state machine after which the current operation is checked against the state of the session type using `_try_advance`. This method compares the given action to that of the session type, raising an exception if it is invalid. Continuing, the message is sent using `_send` which is responsible for the low-level communication. As previously described, `_send` is built using Python's sockets and must wait until a connection to the recipient can be made where it will then encode the message and send it.

```python
def send(self, e: Any) -> None:
    actor = self.session_type.outgoing_actor()
    self._try_advance(Action.SEND, e)
    self._send(e, self.rolesToPorts[actor])
    self._close_if_complete()

def _send(self, e: Any, to: tuple[str, int]) -> None:
    with _spawn_socket() as client_socket:
        try:
            self._wait_until_connected_to(client_socket, to)
            payload = _encode((e, self.roles_to_ports['self'])) # serialise message
            client_socket.sendall(struct.pack('>I', len(payload))) # message size
            client_socket.sendall(payload) # message body
        except ...
```

Recv like `send` must first validate the operation against the session type. Given that this holds, a message from the expected recipient will be returned. A naïve implementation of `recv` may simply listen on a socket and wait until another peer connects and accepts a message. This is sufficient for scenarios with only two participants but can fail when more parties are involved. Consider this protocol with three participants:

```
global protocol OutOfOrder(role A, role B, role C) {
    Msg(int) from B to A;
    Msg(int) from C to A;
}
```

The protocol imposes an ordering of the messages; A should first receive a message from B and thereafter C. This protocol is well-typed but in practice it is observed that messages may arrive in an arbitrary order due to the nature of distributed systems. Given this possibility, a mechanism to allow withholding messages until the right recipient is requesting a message must be implemented. To achieve this, a separate thread listens on the designated local port and puts any messages received into a queue for that participant:

```python
def _listen(self):
    self.server_socket.listen()
    while True:
        try:
            if not self.running: # external shutdown flag
                break
            conn, _ = self.server_socket.accept()
            with conn:
                data_size = struct.unpack('>I', conn.recv(4))[0]
                received_payload = b""
                remaining_payload_size = data_size
                while remaining_payload_size != 0: # receive the entire message
                    received_payload += conn.recv(remaining_payload_size)
                    remaining_payload_size = data_size - len(received_payload)
                msg, addr = _decode(received_payload) # deserialise the message
                sender = self.ports_to_roles[addr]

                if sender in self.message_queue: # enqueue if the sender is known
                    self.message_queue[sender].enqueue((msg, sender))
        except ...
```

This allows A to receive the messages in any order but read them in the correct order. If C should happen to send its message before B, C's message will be enqueued and withheld until the message from B has been received. A's message queue after C sends its message but before B's message is received, can be seen below.

```
'B': [] # no message from B yet
'C': [(42, 'C')] # C has sent a message
```

When the `recv` operation is called on an endpoint, the state machine must first validate the current state of the session type as in `send` and given this succeeds, a call to `_recv` is made. `_recv` uses the previously described internal message queue, blocking other participants until the expected recipient tries to retrieve a message, thereby preserving order:

```python
def recv(self) -> Any:
    actor = self.session_type.outgoing_actor()
    self._try_advance(Action.RECV, None)
    message = self._recv(actor)
    self._close_if_complete()
    return message

def _recv(self, sender: str) -> Any:
    try:
        while True:
            queue = self.message_queue[sender]
            if queue.is_empty():
                continue
            return queue.dequeue()[0]   # (msg, role) tuple
    except ...
```

With the `send` and `receive` primitives already defined, implementing `offer` and `choose` is straightforward.

Offer simply piggybacks on the `_recv` operation, expecting to receive a label. As offer and choice are dual, when an offer is called a choice is guaranteed to be on the other side. It can therefore be assumed a label has been sent by the other party and can be received. The state machine should like the other operation always be in the correct state and it should be checked for that here as well:

```python
def offer(self) -> str:
    actor = self.session_type.outgoing_actor()
    label: str = self._recv(actor)
    self._try_advance(Action.BRANCH, label)
    self._close_if_complete()
    return label
```

Choose piggybacks on the `_send` operation, much like `offer` does with receive, and is implemented in the same way, this time sending what is expected to be a label:

```python
def choose(self, label: str) -> None:
    actor = self.session_type.outgoing_actor()
    self._try_advance(Action.BRANCH, label)
    self._send(label, self.rolesToPorts[actor])
    self._close_if_complete()
```

3.2.6 Projecting Scribble Protocols

In this section, the source-to-source compiler from Scribble protocols to runnable Python code is presented. The goal of utilising the Scribble protocol language is to end up with a template Python file that allows the programmer to fill in the implementation details themselves. Given a well-typed global Scribble protocol, a local protocol matching the individual roles should be generated. From each of these local protocols, a corresponding Python file should be generated that is supplied with an endpoint that matches the local protocol. The full projection workflow can be seen in figure 8.

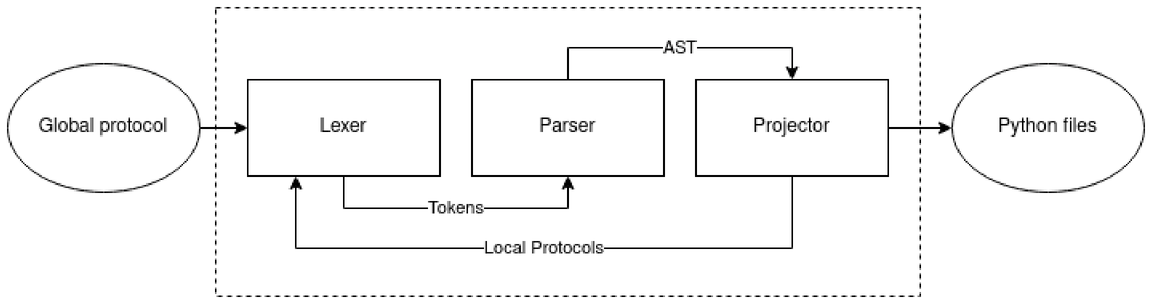


Figure 8: Projection workflow

Extensions made to the Scribble protocols allow for more implementation-specific details that do not reduce the readability of the protocol. These changes include labels on the different branches of choose/offer such that the generated Python file can create a mapping from labels to session types. Choices have also been extended in such a way that it is made explicit who receives the offer and who will make a choice. The extensions can be seen in listing 23, which defines a simple operations server, allowing simple mathematical operations. Each branch has a label that corresponds to the matching operation, and allows the generation of the Python dictionary, mapping a label to a session type {'add': ST, 'neg': ST, 'mul': ST}. In this protocol, it is made clear that the Server receives a choice from the client and allows the state machine to generate the correct transition.

Listing 23 Protocol extensions

```
global protocol OpServer(role Server, role Client) {
    choice from Client to Server {
        @add;
        Num(int) from Client to Server;
        Num(int) from Client to Server;
        Num(int) from Server to Client;
        End;
    } or {
        @neg;
        Num(int) from Client to Server;
        Num(int) from Server to Client;
        End;
    } or {
        @mul;
        Num(int) from Client to Server;
        Num(int) from Client to Server;
        Num(int) from Server to Client;
        End;
    }
}
```

Lexing The first step of the process is breaking down a global protocol into its respective local protocols. Firstly, a series of lexemes should be generated from the protocol. These lexemes are defined based on the Scribble grammar presented in section 2.5 along with the extensions made to it. The tokens consist of strings that can be sequences of alphanumeric characters such as `protocol` or characters with specific meanings in Scribble, such as '{' or ';'. The lexemes defined could for example include `LEFT_PARENS`, `SEMICOLON` or `GLOBAL`. Lexing a file is straightforward, only requiring traversing a file as seen in the `_scan_tokens` method. While there are more tokens to be lexed, the current character is examined using the `_scan_token` method. If the current tokens match any of the single-character tokens it is simply added to the resulting lexeme sequence. Multi-character words can either be keywords or identifiers as seen in the `_identifier` method. The entire string is found and classified as a keyword if there exists a mapping from the string to one of the predefined keywords, otherwise, it is classified as an identifier. Lastly, the EOF terminator is added to the lexeme sequence.

```python
def _scan_tokens(self) -> None:
    while not self._is_at_end():
        self.start = self.current
        self._scan_token()

    self._add_token(TokenType.EOF)

def _scan_token(self) -> None:
    match self._advance():
        case '(':
            self._add_token(TokenType.LEFT_PARENS)
        case ...
        case x if x in [' ', '\r', '\t', '\n']:
            return
        case x:
            self._identifier()


def _identifier(self) -> None:
    while _is_alphanumeric(self._peek()):
        self._advance()

    text: str = self.source[self.start:self.current]
    if _is_keyword(text):
        self._add_token(keywords[text])
    else:
        self._add_token_literal(TokenType.IDENTIFIER, text)
```

Parsing Given a sequence of lexemes provided by the lexer, the parser can turn these tokens into an AST. This is done following a specified grammar that dictates how these tokens should fit together. This grammar expands upon Scribble's own with additional extensions. The grammar can be seen in listing 26 in the appendix, and describes both global and local protocols.

The implementation of the parser seeks to follow the grammar as closely as possible. For each rule, a class is defined and contains the result of any rule calls with any keywords or tokens discarded. To easily be able to parse these rules, the parser exposes an API that seeks to mimic the tokens and quantifiers of regular expressions: **t** (exactly one), **t r** (t and then r), **t** | **r** (t or r), **t?** (zero or one), **t*** (zero or many) and **t+** (one or many). These operations can be seen on the following table, with the corresponding Python function:

Operation	**Python**
t	_match(t)
$t\ r$	_match(t, r)
$t \mid r$	_or(t, r)
$t?$	zero_or_one(t)
$t*$	many(t)
$t+$	one_or_many(t)

The top `protocol` rule specified as the grammar as `protocol ::= typedef* (P | L)`, should first parse *zero* or *many* `typedef` rules followed by either a `P` *or* `L`. In Python, this is implemented using the previously defined API:

```python
class Protocol:
    def __init__(self, typedefs: List[TypeDef], protocol: P | L):
        self.typedefs = typedefs
        self.protocol = protocol

class TypeDef:
    def __init__(self, typ: Typ, identifier: Identifier):
        self.typ = typ
        self.identifier = identifier

def _protocol(self) -> Protocol:
    typedefs: List[TypeDef] = many(self._typedef)
    protocol: P | L = self._or('P | L', self._p, self._l)
    return Protocol(typedefs, protocol)

def _typedef(self) -> TypeDef:
    self._match(TokenType.TYPE, TokenType.LT)
    typ = self._typ()
    self._match(TokenType.GT, TokenType.AS)
    identifier = self._identifier()
    self._match(TokenType.SEMICOLON)
    return TypeDef(typ, identifier)
```

Passing the sequence of tokens generated by lexing the two-buyers protocol to the parser, results in the AST that be seen in figure 9 with a root `Protocol` node with some type definition and the global protocol `G`, which in turn has a name, several roles, and multiple statements (4 `GlobalInteraction` and a `GlobalChoice` node), which each contains related information about the sender/receiver and payload.

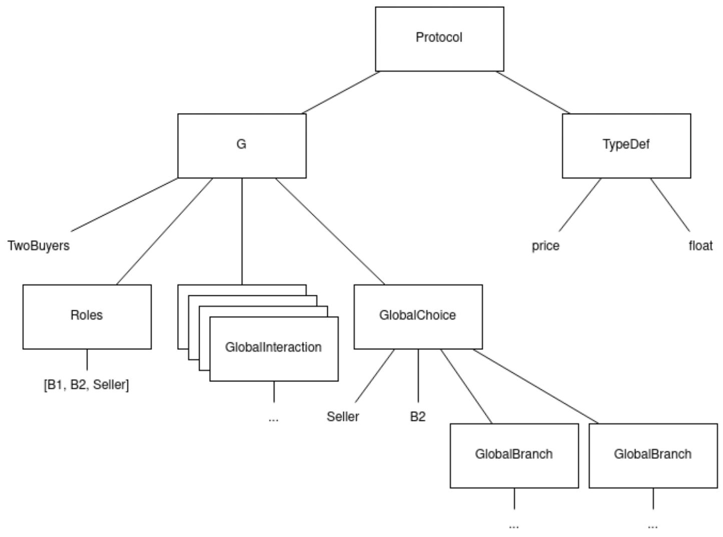


Figure 9: Part of the abstract syntax tree generated by parsing the two-buyers protocol

Projecting Using the previously generated AST of a global protocol, it is possible to project it into a local protocol for each of the participating roles. The projected local protocol for the *Seller* can be seen in listing 24.

Listing 24 Generated local protocols

```
# seller.scr
local protocol TwoBuyers at Seller(role B1, role B2, role Seller) {
    Title(str) from B1;
    Quote(price) to B1;
    Quote(price) to B2;
    choice from B2 {
        @buy;
        Address(str) from B2;
        End;
    } or {
        @reject;
        End;
    }
}
```

The work is done by the *projector* that is implemented much like an interpreter that recursively traverses the AST until every node has been visited starting from the top. In this case the root `Protocol` node is first visited, followed by the `TypeDef` node, and then `G` that is recursively projected. Figure 10 gives a high-level overview of the translation process from firstly the global protocol (left column), to local protocols (middle column) to finally Python code in the right-most column. As per the first entry, to project a `G` AST node to the corresponding `L` AST node, the text 'global protocol X (A, B, C) {Y}' creates three files that each contain 'local protocol X at *role name* (A, B, C) {Y}' where X indicates the protocol name, A, B and C are roles, and Y is a series of statements that are recursively projected.

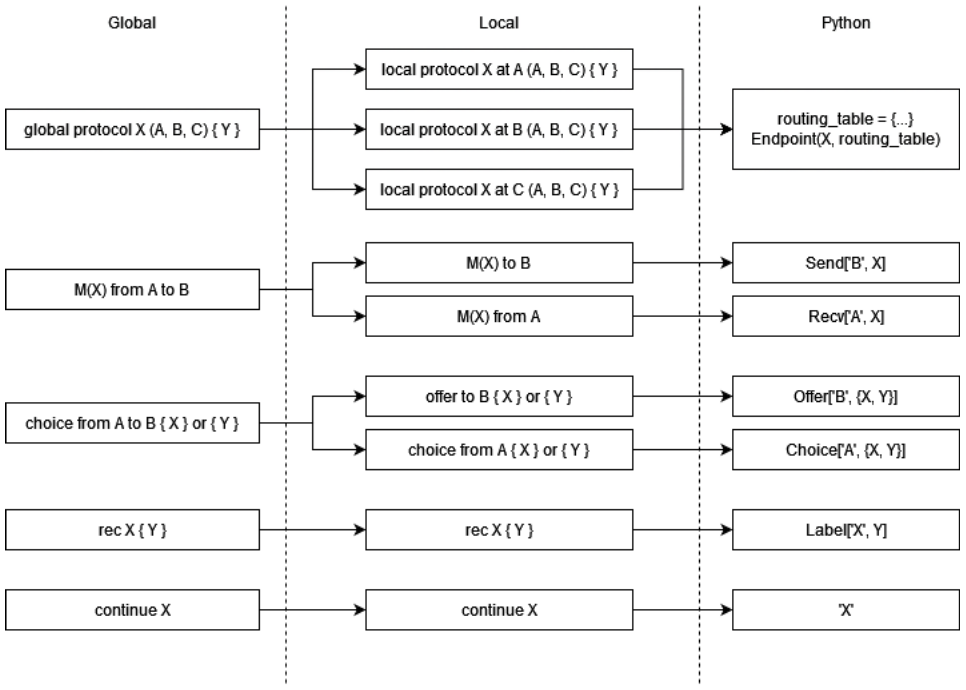

Global
Local
Python
global protocol X (A, B, C) { Y }
local protocol X at A (A, B, C) { Y }
local protocol X at B (A, B, C) { Y }
local protocol X at C (A, B, C) { Y }
routing_table = {...}
Endpoint(X, routing_table)
M(X) from A to B
M(X) to B
M(X) from A
Send['B', X]
Recv['A', X]
choice from A to B { X } or { Y }
offer to B { X } or { Y }
choice from A { X } or { Y }
Offer['B', {X, Y}]
Choice['A', {X, Y}]
rec X { Y }
rec X { Y }
Label['X', Y]
continue X
continue X
'X'

Figure 10: Projection flow

A similar process has been implemented for local protocols, that creates the corresponding Python file from a local protocol. The created Python file contains a `routing_table` that contains information about the addresses of each role to correctly send the messages to the right role, and the local protocol translated into a session type that is used in the endpoint. The Python file generated based on the *Seller* local protocl can be seen in listing 25

Listing 25 Generated Python files

```
routing_table = { ... } # addresses omitted

ep = Endpoint(Recv[str, 'B1', Send[float, 'B1', Send[float, 'B2', Offer['B2',
↪  {"buy": Recv[str, 'B2', End], "reject": End}]]]], routing_table)
```

4 Evaluation

This section presents the results of using SessionPy to implement different programs in section 4.1. SessionPy is then compared to other related implementations in section 4.2, concluding with a discussion of the general usability and philosophy of session types, along with potential future work in section 4.3.

4.1 Example Programs

This section presents four example implementations:

1. A distributed MapReduce example
2. The two-buyers protocol
3. The DynServer example
4. An interruptible stream protocol

MapReduce is a programming model for big data processing that is often used among several processes or systems and is a good way to highlight the distributed aspect of SessionPy. A model of a distributed word count program can be implemented using the MapReduce technique first by writing a simple protocol that from a central server—the *distributor*—will send a list of words that should be counted to some working machines $\{c_1, ..., c_3\}$. Each of the worker machines will then perform the *map* operation (word count) on the received workload and send back a result. The `distributor` will then perform the *reduce* operation and combine all the results. The global protocol can be seen below. It firstly defines two types as seen on line 1 and 2: *workload* representing the list of words that should be counted, that is mapped to the Python *list[str]* data type. *result* is the result from the worker machines back to the `distributor` and is represented as a Python dictionary mapping strings to integers. The protocol is then specified from line 4 and onwards, declaring a protocol called WordCount with the four roles previously mentioned. The distributor then sends its payload to each of the worker machines (l. 5 - 7) and then receives the results (l. 9 - 11) and finally stops the protocol.

```
1  type <list[str]> as workload;
2  type <dict[str, int]> as result;
3
4  global protocol WordCount(role Distributor, role c1, role c2, role c3) {
5      Work(workload) from Distributor to c1;
6      Work(workload) from Distributor to c2;
7      Work(workload) from Distributor to c3;
8
9      Res(result) from c1 to Distributor;
10     Res(result) from c2 to Distributor;
11     Res(result) from c3 to Distributor;
12     End;
13 }
```

Projecting the global protocol then results in four local protocols, one for each of the workers and one for the distributor. This can be seen below where firstly the distributor's protocol is listed (l. 1 - 10) and then the worker's protocol (l. 13 - 17).

```
1  # distributor.scr
2  local protocol WordCount at Distributor(role Distributor, role c1, role c2, role
   ↪  c3) {
    Work(workload) to c1;
    Work(workload) to c2;
    Work(workload) to c3;
    Res(result) from c1;
    Res(result) from c2;
    Res(result) from c3;
    End;
}

# {c1, c2, c3}.scr
local protocol WordCount at c1(role Distributor, role c1, role c2, role c3) {
    Work(workload) from Distributor;
    Res(result) to Distributor;
    End;
}
```

Projecting each local protocol, the corresponding Python file is generated. The implementation of the distributor (l. 1 - 19) sends a list of words that should be counted to each of the workers (l. 4 - 6) and receives their result once they are done (l. 8 - 10). Hereafter the dictionaries are merged into one (l. 12 - 17) and lastly outputting the collected list (l. 19) completing the word count from the distributor's point of view. Each worker machine (l. 21 - 31) starts off by receiving its designated workload (l. 24), tallies them (l. 28, 29), and sends the result back (l. 31).

```
1  # distributor.py
2  ep = Endpoint(Send[list[str], 'c1', Send[list[str], 'c2', Send[list[str], 'c3',
   ↪  Recv[dict[str, int], 'c1', Recv[dict[str, int], 'c2', Recv[dict[str, int],
   ↪  'c3', End]]]]]], routing_table, static_check=False)
3
4  ep.send(['Deer', 'Bear', 'River', 'River', 'Bear'])
5  ep.send(['Car', 'Car', 'River', 'River', 'River'])
6  ep.send(['Deer', 'Car', 'Bear', 'Car'])
7
8  r1: dict[str, int] = ep.recv()
9  r2: dict[str, int] = ep.recv()
10 r3: dict[str, int] = ep.recv()
11
12 # merge dictionaries
13 for k, v in r2.items():
14     r1[k] = r1.get(k, 0) + v
15
16 for k, v in r3.items():
17     r1[k] = r1.get(k, 0) + v
18
19 print(r1) # {'Deer': 2, 'Bear': 3, 'River': 5, 'Car': 4}
20
21 # {c1, c2, c3}.py
22 ep = Endpoint(Recv[list[str], 'Distributor', Send[dict[str, int], 'Distributor',
   ↪  End]], routing_table, static_check=False)
23
24 words: list[str] = ep.recv()
25
26 result: dict[str, int] = {}
```

```python

for w in words:
    result[w] = result.get(w, 0) + 1

ep.send(result)
```

This example demonstrates a practical use case of the endpoint abstraction and how it is integrated with session types. To implement a distributed word count example, only the logic of the application should be implemented along with some initial routing details. While this example is very simple, and does not use any branching session types, the next example highlights how these are handled.

Two Buyers The next example is the two buyers protocol also mentioned in section 2.3. The global protocol is projected into the respective local protocols (listing 24). Hereafter the stub Python files that were generated (listing 25) have been extended following the original specifications. Firstly, the implementation for b_1 is shown on lines 1 - 6. As specified by the session type they must first send the title to a book to the seller, which in this case is 'War and Peace' (l. 4) whereafter it receives a quote from the seller and sends half of that quote to $b2$ (l. 5, 6). b_2 firstly receives the quote from the seller (l. 11) and b_1's contribution (l. 12). b_2 then makes a choice based on the quote and b_1's contribution to either buy the book or reject it (l. 14). Is the book affordable, b_2 will choose the 'buy' option and send their address to the seller whereafter the protocol is done (l. 15, 16). Otherwise, they will simply reject the purchase and end the protocol (l. 17). The seller firstly receives the book title from b_1 (l. 23) whereafter it proceeds to send a quote to both b_1 and b_2 (l. 26, 27). Hereafter it waits for an answer from b_2 as seen on line 29, where both cases are handled following the session type.

```python
1  # b1.py
2  ep = Endpoint(Send[str, 'Seller', Recv[float, 'Seller', Send[float, 'B2', End]]],
   ↪  routing_table)
3
4  ep.send('War and Peace')
5  quote = ep.recv()
6  ep.send(quote / 2)
7
8  # b2.py
9  ep = Endpoint(Recv[float, 'Seller', Recv[float, 'B1', Choose['Seller', {'buy':
   ↪  Send[str, 'Seller', End], 'reject': End}]]], routing_table)
10
11 quote = ep.recv()
12 contrib = ep.recv()
13
14 if quote - contrib <= 99:
15     ep.choose('buy')
16     ep.send('here is my address')
17 else:
18     ep.choose('reject')
19
20 # seller.py
21 ep = Endpoint(Recv[str, 'B1', Send[float, 'B1', Send[float, 'B2', Offer['B2',
   ↪  {'buy': Recv[str, 'B2', End], 'reject': End}]]]], routing_table)
22
23 title = ep.recv()
24
25 quote = 120.0
```

```
ep.send(quote)
ep.send(quote)

match ep.offer():
    case 'buy':
        address = ep.recv()
    case 'reject':
        ...
```

This example highlights SessionPy's ability to handle branching in protocols as is done by the `match` statement performed by the seller, and the `choose` operation performed by b_2.

Compute Service Next, the Compute Service example as presented in section 2.3.3 is shown. As mentioned, this example consists of two parts: Sy wanting to write the client using static typing and Rob wanting to use dynamic typing to write the server. This example highlights the gradual aspect of SessionPy. A global protocol describing the compute service is firstly written and projected into a local protocol for both. The local protocols have been omitted. The protocol is very simple, simply a choice between a negation or addition operation that Sy should make.

```
type <int> as n;

global protocol ComputeService(role Server, role Client) {
    choice from Client to Server {
        @neg;
        Num(n) from Client to Server;
        Num(n) from Server to Client;
        End;
    } or {
        @add;
        Num(n) from Client to Server;
        Num(n) from Client to Server;
        Num(n) from Server to Client;
        End;
    }
}
```

The implementation-specific typing details are controlled by a named argument `static_check` which toggles the static checking of the file. This allows almost a 1:1 translation of the Gradual GV example into SessionPy and allows for both parts to be written with their desired style.

```
# client.py
session_type = Choose['Server', {'neg': Send[int, 'Server', Recv[int, 'Server',
↪  End]],
                                  'add': Send[int, 'Server', Send[int, 'Server',
                                  ↪  Recv[int, 'Server', End]]]}]
ep = Endpoint(session_type, routing_table)


def do_negate(c: session_type):
    c.choose('neg')
    c.send(42)
    print(c.recv())
```

```


def do_add(c: Send[int, 'Server', ...]):
    c.send(42)
    c.send(42)
    print(c.recv())


do_negate(ep)

# server.py
ep = Endpoint(Offer['Client', {'neg': Recv[int, 'Client', Send[int, 'Client',
↪  End]], 'add': Recv[int, 'Client', Recv[int, 'Client', Send[int, 'Client',
↪  End]]]}], routing_table, static_check=False)

negate = lambda x: -x
add = lambda x: lambda y: x + y


def dyn_server(c):
    match c.offer():
        case 'neg':
            serve_op(1, negate, c)
        case 'add':
            serve_op(2, add, c)


def serve_op(n, op, c):
    if n == 0:
        c.send(op)
    else:
        v = c.recv()
        serve_op(n - 1, op(v), c)


dyn_server(ep)
```

Interruptible Stream Finally, a protocol representing an interruptible stream of integers has been implemented to demonstrate the recursive case of SessionPy. This protocol continuously sends integers from a Source to a Sink, until at some point the stream will stop. Again, the local protocols have been omitted. The body of the protocol is wrapped in a `rec` block with a *LOOP* label, indicating the protocol can be repeated. The body of the loop is then a choice to either send a message and continue the loop (@work) or terminate (@stop).

```
global protocol Stream(role Source, role Sink) {
    rec LOOP {
        choice from Source to Sink {
            @work;
            Send(int) from Source to Sink;
            continue LOOP;
        } or {
            @stop;
```

```
                End;
            }
        }
    }
```

The implementation of the source governs how long the stream should run. In the implementation, this is controlled by the i variable, stopping when it becomes too large (l. 7). While i is less than some amount, the 'work' branch is continuously chosen, and messages are sent until i exceeds some value (l. 8, 9). As the protocol is marked as potentially repeating, it should be put in a loop to allow for repetition. Likewise, the implementation of the sink is also in a loop and at each iteration, matches on whatever choice the source may take (l. 19). Running the protocol, the sink will print the integers 0 to 199.

```
1   # source.py
2   ep = Endpoint(Label['LOOP', Choose['Sink', {'work': Send[int, 'Sink', 'LOOP'],
    ↪  'stop': End}]], routing_table)
3
4   i = 0
5
6   while True:
7       if i < 200:
8           ep.choose('work')
9           ep.send(i)
10      else:
11          break
12      i += 1
13
14  ep.choose('stop')
15
16  # sink.py
17  ep = Endpoint(Label['LOOP', Offer['Source', {'work': Recv[int, 'Source', 'LOOP'],
    ↪  'stop': End}]], routing_table)
18
19  while True:
20      match ep.offer():
21          case 'work':
22              print(ep.recv())
23          case 'stop':
24              break
```

4.2 Related Work

Gradual Session Types This paper by Igarashi, Thiemann et al. presented in 2019 is some of the most thorough work in bridging hybrid type systems with session types we discovered. To solve the problem of allowing communication between code fragments with varying typing discipline, they extend an existing language, GV, with two new typing constructs; the dynamic type—corresponding to the Any type in Python—and the dynamic session type that represents all possible session types. This novel extension is to give home for dynamism within a static type system. Functions can then in return be typed using these new dynamic types, allowing for basic types and session types to vary. Part of the compilation phase will then cast functions, primitives, and session types into dynamic types, and in the receiving function cast these back into the expected types to type check the program. Where these casts happen so-called blame labels are inserted during compilation to keep track of where the responsibility lies if during runtime these casts should fail. Gradual GV (GGV), the language presented in the paper, has several similarities to Python. They both allow for dynamic and static typing, and everything in between. The star type representing the dynamic type in GGV is equivalent to the `Any` type in Python, and they both allow for explicit function signatures. The main difference is that whereas GGV is a gradual language, Python is entirely dynamic, and does not respect any optionally provided typing information. Much of the work on SessionPy was to make it respect this provided information statically. We did not fully adopt the concepts of blame labels and the cast-related method they impose to handle gradual session types. Implementing their idea of casting session types was found to be infeasible due to Python's type-neglecting nature but instead inspired the stub notation of session types used for statically annotating functions with partial session types. Similarly, it highlighted the importance of creating a seamless transition between handling session types in both the static and dynamic type paradigm that does not impose any extra workload on the end user.

Session Types for Rust In 2015, Munksgaard et al. presented their work on binary session types implemented in Rust. The Rust language is renowned for its affine type system that statically keeps track of ownership of variables using its borrow system and move semantics. Dubbed a *safe* language (yes, it even has an `unsafe` keyword for writing C-style code), these semantics provide certain guarantees about memory safety and avoidance of race conditions when writing concurrent programs. This is also part of the reasons for its fit with session types; the linearity of Rust's type system resembles the nature of linearity of session types. In the paper, the type system of Rust is being leveraged to create structures representing each session typing construct; send, receive, branching, recursion, and epsilon representing inaction. While Python and Rust are two very different programming languages, the work on recursion in this paper, especially the idea of using peano numbers to index which protocol to be repeated inspired the labelling system for recursion. Not being constrained by any existing type system, labelling—instead of indexing—using strings resulted in a more pythonic solution.

Practical Interruptible Conversations Where Munksgaard et. al. deal with static checks we mainly got our ideas on dynamic verification from the paper by Hu, Neykova et. al. that presents an API for conversation programming in Python. A *conversation* refers to a communication session between two or more participants, making it interchangeable with simple *sessions* in our project. The paper from 2013 was written based on a collaboration with industry partner Ocean Observatories Initiative (OOI), and focuses on dynamic verification using a runtime monitor. Much work up until this point in time has focused on static verification of session types; verifying during compile-time that the conversation adhered to the protocol. The Conversation API resembles the constructs known from MPST theory, such as the send and receive operations, and are written directly in plain Python code by using the conversation API. At each endpoint of the conversation, a finite state machine (FSM) is generated which is used as part of the runtime monitoring. This monitor acts as an intermediary between endpoints to verify the communication is adhering to the protocol specification. The combination of Scribble and Python for SessionPy is largely inspired by this project, as well as having an idea of a finite-state machine to progress in a type, but where they utilise an external monitor, we leverage on our gradual typing system for statically verifying some parts, and then using runtime checks at each endpoint for the remainder.

4.3 Discussion

As demonstrated in section 4.1, SessionPy proves to be expressive enough to model and implement smaller distributed systems along with selected case-study protocols proposed by other papers, such as the two-buyers and compute service example. It supports inter-process communication and implements all the standard session functionality such as message passing, branching, and recursion. In addition, static checking proves helpful during development in ensuring the implementation matches the session type.

All the guarantees of session types are upheld by SessionPy. The combined implementation of endpoints, type checker and session types guarantee communication safety when it is statically checked, linearity as a single operation is guaranteed to only be used once, deadlock freedom under the assumption the implemented protocol is well-typed, and lastly session fidelity as the specified session type is always followed. One feature that may be lacking, is the handling of any errors. As networks are unstable by default, messages may fail to arrive or participants can lose connection during the conversation.

Static and dynamic type systems both have well-defined strengths and weaknesses. A safety-critical system will arguably benefit from a static type system and compile-time errors, whereas smaller, fast-paced projects with shorter lifespans might be a perfect fit in a dynamic language. Gradual typing is conceptually a slider between the two; it allows the programmer added flexibility without forcing them to switch language. Having to write an external static type checker to implement this gradual type system felt counter-intuitive working with Python which is an interpreted language even with all the new features introduced by PEP 484. Another option could therefore have been to use an existing type checker for Python, and extend it with session types. This would however not be trivial, as the current implementation of session types bends some of Python's rules such as the stub notation. Likewise, some of the external type checkers are only for previous Python versions that do not support some of the features used in the implementation of SessionPy.

With the goal of session types being to implement correct and verifiable communication, a strong type system is greatly desired. With Python having a very weak type system, session types may seem like an odd addition as they are so reliant on types and static checking. An argument could therefore be made that fitting such a type-dependent concept to any dynamic language makes little sense and would be a better fit for languages that have a type checker already implemented. Implementing session types for a gradual language like *Dart* or any language that has some low-level network communication could therefore be an interesting idea.

In its current state, SessionPy can only type check single files as supporting imports would require it to visit an exponential amount of other files (say, each visited file imports code from 5-10 other files). A remedy could be to postpone type checking until necessary (lazy evaluation) by introducing a dependency graph on types in SessionPy; by statically keeping track of types and maintaining a graph of what values are sent and received by the endpoints, we would only have to type check exactly those values to validate the session type. Consider the following program.

```
x = 42
y = <complex computation>
ep.send(x)
```

As-is, we use time to validate and typecheck the complex computation, and store the type of `y` despite statically we can argue that it would never affect neither the endpoint nor the type of `x`.

4.4 Future Work

DSL The implemented DSL that is based on Scribble, does not support any checks for well-formedness. This can result in the generation of protocols that does not translate to a valid session type. Likewise, some protocols are not able to be expressed using the current implementation of session types. More specifically, this concerns protocols that perform branching followed by more operations. An obvious extension would therefore be to either integrate SessionPy with the already existing version of Scribble, or to extend the DSL workflow to include a validation step to ensure the protocols are valid before they are projected.

TypeChecker The implemented type checker is eager and will check the entire file and all computations which could be avoided as discussed in 4.3. Future work could therefore include a lazy type checker that builds a dependency-graph, and only check expressions that are related to session types. Similarly, only a subset of Python is checked statically. An obvious extension would therefore be to cover all of Python's AST to fully incorporate session types with the language and extend the capabilities of the static checker. It was discovered that all Python functions have a `__code__` attribute representing code objects. These code objects contain various data from variable information to executable bytecode for the function. Using decorators that allow meta-inspection of functions combined with these code objects could allow us to directly inspect a function and its associated attributes. It remains unclear to what extent we could extract any useful information out of these attributes and validate session types this way, but it could potentially avoid parsing and AST inspection of source files.

Language Semantics Working on SessionPy, much of the time was spent on researching Python's type constructs introduced with PEP484 and doing incremental extensions to the type checker. This also meant that several semantics, such as the validation of if-then-else statements, came up ad hoc during development. For this reason, several language constructs do not have formal operational semantics defined in terms of session types, which could be extended upon as future work. This resulted in the incorporation of the previously defined Python subset to limit the scope of the project and to do proof-of-concept implementation on the feasibility of implementing a type checker and session types for Python and validate them accordingly.

5 Conclusion

In this paper, gradual session types have been implemented in Python by firstly designing a concise way of expressing session types; a type system for describing communication between two or more parties that support sending and receiving values, branching and recursion. Secondly, an endpoint abstraction has been implemented that supports inter-process communication using session types and Python sockets. A gradual type checker has been developed using state of the art typing features in Python that allows for both static and dynamic verification of endpoints. In addition, an external DSL for specifying protocols has been developed that is heavily inspired by Scribble—an existing protocol description language. Programs written in this DSL describe a communication sequence and can be projected into a Python program for each participant with the corresponding session type added as an endpoint. Static inspection and validation of session types happens upon instantiating endpoints, but can also be enforced on functions using decorators. The gradual typing aspect allows the programmer to be more or less specific about the typing information provided, allowing for flexibility in how to write the programs. Future work could include formalising the operational semantics of SessionPy, extending the gradual type checker, or even migrating the implementation to another language with better support of both static and dynamic typing to take full advantage of the gradual aspect of session types.

References

[1] Number of internet of things (iot) connected devices worldwide from 2019 to 2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, accessed: Apr. 26, 2022

[2] Python typing documentation. https://docs.python.org/3/library/typing.html, accessed: Apr. 28, 2022

[3] What is gradual typing. https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/, accessed: Apr. 28, 2022

[4] ATSUSHI IGARASHI, PETER THIEMANN, Y.T.V.T.V.P.W.: Gradual session types (2019)

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

[6] Neykova, R.: Session types go dynamic or how to verify your python conversations (2013)

[7] Pierce, B.C.: Types and programming languages (2002)

[8] Raymond Hu, Rumyana Neykova, N.Y.R.D., Honda, K.: Practical interruptible conversations (2013)

[9] Rumyana Neykova, N.Y.: Featherweight scribble (2019)

A Appendix

Table 1: AST nodes visited during static checks and their syntactic form

AST node	Description	Syntax
AnnAssign	annotated assign	x : y = z
Assign	assign	x = y
Attribute	attribute access	x.y
AugAssign	compound assignment	x += y
BinOp	binary operator	x <binop>y
Break	break statement	break
Call	function call	x(y)
ClassDef	class definition	class x: <stmt>
Compare	comparison	x <compare-op>y
Constant	literal value	i.e. string, integer or None
Dict	dictionary	{x: y, w: z}
For	for-loop	for x in xs: <stmt>
FunctionDef	function definition	def f(x, y): <stmt>
If	if-statement	if <cond>: <stmt> (also, elif and else)
JoinedStr:	formatted strings	f"Hello, {x}!"
Lambda	lambda expressions	lambda x: <stmt>
List	list	[x, y, z]
Match	match statement	match x: <cases>
MatchAs	named match case	case x as y: <stmt>
Name	variable name	<var>
Return	return statement	return <expr>?
Subscript	subscript notation	x[y]
Tuple	tuple (or list)	(x,y,z)
UnaryOp	unary operator	-x
While	while loop	while <cond>: <stmt>

Listing 26 SessionPy-Scribble grammar

```
⟨protocol⟩ ::= typedef* (P | L)

⟨P⟩        ::= global protocol identifier ( roles ) { G* }

⟨G⟩        ::= global_interaction | global_choice | global_recursion | end | call

⟨global_interaction⟩ ::= message from identifier to identifier ;

⟨global_choice⟩ ::= choice from identifier to identifier global_branch (or global_branch)+

⟨global_branch⟩ ::= { branch_label G* }

⟨global_recursion⟩ ::= rec identifier { G* }

⟨L⟩        ::= local protocol identifier at identifier ( roles ) { T* }

⟨T⟩        ::= local_send | local_recv | local_offer | local_choice | local_recursion | end | call

⟨local_send⟩ ::= message to identifier ;

⟨local_recv⟩ ::= message from identifier ;

⟨local_choice⟩ ::= choice from identifier local_branch (or local_branch)+

⟨local_offer⟩ ::= offer to identifier local_branch (or local_branch)+

⟨local_branch⟩ ::= { branch_label T* }

⟨local_recursion⟩ ::= rec identifier { T* }

⟨branch_label⟩ ::= @ identifier ;

⟨message⟩ ::= identifier ( typ );

⟨typedef⟩ ::= type < typ > as identifier ;

⟨roles⟩    ::= role (, role)+

⟨role⟩     ::= role identifier

⟨call⟩     ::= continue identifier ;

⟨typ⟩      ::= identifier ( [ identifier (, identifier)* ] )?

⟨identifier⟩ ::= [A-Z] ([A-Z] | [a-z] | 0 - 9)*

⟨end⟩      ::= End ;
```