A Session Type Provider

Compile-Time API Generation of Distributed Protocols with Refinements in F#

Rumyana Neykova
Imperial College London
United Kingdom

Raymond Hu
Imperial College London
United Kingdom

Nobuko Yoshida
Imperial College London
United Kingdom

Fahd Abdeljallal
Imperial College London
United Kingdom

Abstract

We present a library for the specification and implementation of distributed protocols in native F# (and other .NET languages) based on multiparty session types (MPST). There are two main contributions. Our library is the first practical development of MPST to support what we refer to as *interaction refinements*: a collection of features related to the refinement of *protocols*, such as message-type refinements (value constraints) and message-value dependent control flow. A well-typed endpoint program using our library is guaranteed to perform only compliant session I/O actions w.r.t. to the refined protocol, up to premature termination.

Second, our library is developed as a session *type provider*, that performs on-demand compile-time protocol validation and generation of protocol-specific .NET types for users writing the distributed endpoint programs. It is implemented by extending and integrating Scribble (a toolchain for MPST) with an SMT solver into the type providers framework. The safety guarantees are achieved by a combination of static type checking of the generated types for messages and I/O operations, correctness by construction from code generation, and automated inlining of assertions.

CCS Concepts • **Software and its engineering** → **Source code generation**; • **Computing methodologies** → **Distributed programming languages**; • **Networks** → *Peer-to-peer protocols*;

Keywords Multiparty Session Types, Distributed Programming, F#, Type Providers, Code Generation

ACM Reference Format:
Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-Time API Generation of Distributed Protocols with Refinements in F#. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179495

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CC'18, February 24–25, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5644-2/18/02.
https://doi.org/10.1145/3178372.3179495

1 Introduction

Type providers [20, 27] are a .NET feature for a form of compile-time meta programming, designed to bridge between programming in statically typed languages such as F# and C#, and working with so-called *information spaces*—structured data sources such as SQL databases or XML data.

A type provider works as a compiler plugin that performs on-demand generation of *types*: it takes a schema for an external information space, and generates types that allow the data to be manipulated via a strongly-typed interface, with benefits such as static error detection and IDE auto-completion. For example, an instantiation of the in-built type provider for WSDL Web services [6] may look like

```
// Assume URL gives WSDL description for operation MyOp: int → bool
type svc = Microsoft.FSharp.Data.TypeProvider.WsdlService<"http:...">
```

where the URL points to the WSDL XML document, and `svc` will house the types generated for the client-side service interface as nested types. A client program could then be:

```
let client = svc.GetMyServiceSoap()
let req = new svc.ServiceTypes.myhost.com.Request(myInt = 123)
let res = client.MyOp(req)
printfn "%b" (res.myBool)
```

The various generated types and operations ensure, e.g., that the payload of `Request` is an `int`, that `MyOp` is correctly invoked with a `Request`, and `res.myBool` is a `bool`.

Type providers have proved a popular and valuable tool in the F# community within their primary use case of working with structured *data*—the main type providers library (F# Data [10]) is one of the most downloaded F# libraries [20]. However, type providers have untapped potential in typed code generation and compiler support for important applications beyond data protocols—distributed programming in particular. The WSDL type provider is for Web services, but is limited to handling the *client*-side data of the hardcoded *call-return* pattern. A generalisation to distributed protocols requires a notion of schema for *structured interactions* between multiple participants, and an understanding of how to extract the localised behaviour for any one of them—precisely the key concepts of multiparty session types (MPST).

Session types is a types-based approach to specifying and verifying protocols for concurrent, message passing processes [12]. We can outline the key concepts using the simple protocol from above. The protocol may be *specified* along the lines of:

```
(int) from C to S; (bool) from S to C;
```

This is a *type* that says: *role* (i.e., endpoint) `C` sends an `int` message to `S`, followed by `S` sending back a `bool` message. We are using the syntax of Scribble [24], a toolchain based on *multiparty* session types (MPST) [5, 13]. Then, the above type may be used to statically type check an *implementation* of the communication session, comprised of the concurrent *processes* playing each endpoint. For example, we could write a function `runS` for the `S` endpoint:

```
runS s = let (x, s') = receive s in send true s'
```

Here we are using the theoretical functional language referred to as GV [11] informally for this initial illustration. Assuming `s` is a freshly initialised channel for `S` to communicate with `C`, `runS` may be considered *well-typed* according to the above type: `s` is first used to receive the `int` from `C` (bound to `x`), with the continuation of the protocol to be performed on the channel bound to `s'`—on which we send `true`. Protocol violations such as attempting to use `send` on `s`, or to send `123` on `s'`, would be caught as type errors.

The main safety properties of session types are that a well-typed system of endpoint processes is guaranteed free of critical communication errors such as *reception errors* (unexpected messages), *deadlocks* (wait-for cycles between processes due to mutual input dependencies) and *orphan messages* (leftover messages). These properties are sometimes referred to as *communication safety* [13]. The theoretical developments have lead to active research on implementations of languages and tools based on session types; see [1, 3] for a broad overview.

So far, the state of the art has primarily focused on implementing the features of the core theory. One of the most promising directions for improving the applicability of session types is the incorporation of concepts from *refinement types* [9, 17]. In a nutshell, the basic notion of refinement type is a data type elaborated with a logical constraint; e.g., in the function type `sqrt: T → int`, the parameter `T` could be the refinement type $\{x : \texttt{int} \mid x >= 0\}$. In the distributed programming setting of session types, there are many ways in which it would be useful to refine a *protocol*; e.g., refinements on the communicated message-types, message-value dependent control flow, and assertions on session state. We shall collectively refer to such features as *interaction refinements*, aspects of which have been studied across a range of separate theoretical systems, e.g. [2, 4, 29]. To date, there has been no work on formulating or implementing session types with such refinements in practice.

Contributions. This paper presents a practical library for MPST-based distributed programming in F# using type providers. The main contributions are as follows.

- We leverage the type provider mechanism for on-demand generation of types to support MPST-based programming in native F#. From MPST, our *session type provider* (STP) promotes static prevention of protocol violations via the type checking of programs using the generated types. From type providers, we obtain benefits [27] such as avoiding workflow bloat (no additional preprocessing steps or external tool stages) and robustness (automated maintenance of generated code against user programs).
- The STP is the first implementation of session types that supports *interaction refinements*. We develop a practical validation method for our enriched protocol specifications by integrating an extension of Scribble and an SMT solver (Z3 [30]) into the type provider framework.
- We build on a hybrid code generation approach [14], further exploiting *type-driven code generation* to safely enforce certain value-dependent communication patterns in user programs *by construction*.

As a taster, the STP allows the specification of communication patterns (written in our extended Scribble) like:

(i) `(x:int) from A to B; @x>0` `(x) from B to C;`

The first interaction (left) uses a *message-type refinement* to specify that `A` must send an `int` greater than zero. The following interaction expresses a *message-value dependency*, specifying that `B` must send exactly the received value to `C`.

As an example of protocol validation, the STP will determine that the above is a valid protocol; unlike, e.g.,

```
(x:int) from A to B; @x>0 (y:int) from B to C; @(y>x and y<5)
```

which would unsafely allow `A` to send an `x` value (e.g., 5) that makes it impossible for `B` to produce a valid `y`.

Given a valid protocol, the STP can *enforce* it in the user program in various ways. For example, a minimal F# fragment for the `B` endpoint in (i) above would be

```
s.Receive(A, x).Send(C)
```

where `s` is `B`'s initial session channel and `x` is a buffer for receiving the `int`. Firstly, the type provider generates the type of `s` to permit *only* the `Receive` from `A`, whose generated return type is the *protocol continuation* permitting only the `Send` to `C`. Secondly, the `Send` has *no* `int` parameter, because the type provider generates it to implicitly send the previously received value, satisfying the refinement *by construction*. In other situations, the type provider may automatically inline a refinement expression as a run-time assertion into the underlying code, as a default enforcement mechanism.

The type providers benefit of avoiding user workflow bloat is important. While it is a challenge to develop advanced programming tools, it is also often a challenge for end users to incorporate them into their regular development environments and actually use them. Our approach exercises the type providers framework as a platform for tools integration—Scribble, Z3 and .NET code generation—making session types (with interaction refinements) practically accessible to .NET programmers unfamiliar with these techniques.

Outline. We start with an overview of our framework in § 2, and then explain the key stages in § 3–5. Compile-time and run-time performance is evaluated in § 6. The source

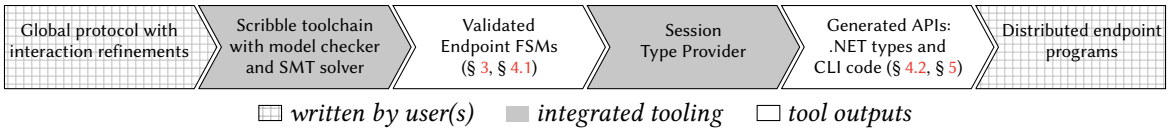


Figure 1. Session type provider toolchain: *compile*-time API generation from MPSTs with interaction refinements.

code of the STP, example applications and omitted details are available from [26].

2 Overview

This section gives an overview of using the *session type provider* (STP) for protocol-driven distributed programming in F# (cf., schema-driven data type providers), with support for interaction refinements. Fig. 1 shows the user inputs and main components of our toolchain and their dependencies, which we illustrate by the following running example.

The Sutherland-Hodgman algorithm (henceforth, SH) is for polygon clipping. It takes a plane, and the vertices of a polygon as a series of points; and produces vertices for the polygon restricted to one side of the plane. We describe a variant of a distributed pipeline implementation [21, 25]. There are three main components: the *Producer* of the input polygon data, the *Calculator* for geometric calculations, and the *Consumer* of the clipped output polygon data. The pipeline can be formed by linking Producers and Consumers, one stage for each clipping plane.

The Producer iterates over adjacent vertices of the polygon. It uses Calculator to determine whether or not the points in each edge pair are *above* the clipping plane; and forwards either zero, one or two points to the Consumer. If both points are above, the second is forwarded; if both are below, neither are forwarded. If only one is above, then the *intersection* of their edge with the plane (using Calculator) is forwarded; and if the second point is above, it is also forwarded.

2.1 The Session Type Provider Toolchain

Global protocol specification in Scribble. Starting from the left of Fig. 1, we use SH to demonstrate the specification of communication patterns with refinements in our extension of Scribble. Fig. 2 gives a *global* protocol for SH, which describes all the required and permitted participant interactions from a global perspective. The root protocol `SH` declares the three participant *roles*: the Producer `P`, the Calculator `R` and the Consumer `C`. It starts with `P` sending a `Plane(…)` message to `R`: `Plane` is a message *label* (e.g., an identifier in a header field), and the tuple of `Point` types is the message *payload* (for simplicity, hardcoded to four points). The `do`-statement then enters the `Loop` subprotocol.

In `Loop`, `choice at P` is a branch point in the protocol where `P` decides which of the `or`-separated cases to follow. In each case, the protocol specifies the communication of the decision result by explicit messages to the other roles (leaving

```
1  global protocol SH(role P, role R, role C) {
2    Plane(Point, Point, Point, Point) from P to R;
3    do Loop(P, R, C);
4  }

5  global protocol Loop(role P, role R, role C) {
6    choice at P {
7      IsAbove(v1:Point) from P to R; Res(b1:bool) from R to P;
8      IsAbove(v2:Point) from P to R; Res(b2:bool) from R to P;
9      choice at P {
10       BothIn(v2)       from P to C;  @ (b1 and b2)
11       BothIn()         from P to R;
12     } or {
13       BothOut()        from P to C;  @ not(b1 or b2)
14       BothOut()        from P to R;
15     } or {
16       Intrsct(v1, v2)  from P to R;  @ (b1 xor b2)
17       Res(i:Point)     from R to P;
18       choice at P {
19         SecOut(i)      from P to C;  @ not(b2)
20       } or {
21         SecIn(i, v2)   from P to C;  @ b2
22     } }
23     do Loop(P, R, C); // Recursion
24   } or {
25     Close() from P to R; Close() from P to C; // End
26 } }
```

Figure 2. A distributed Sutherland-Hodgman algorithm.

the concrete decision procedure abstract). In the second case (line 24), `P` sends `R` and `C` a `Close` message (with no payload values), and the protocol ends.

The first case iterates the algorithm one step. `P` uses the `isAbove` function of `R` on the first of the current points in this iteration, and `R` returns the boolean result; similarly for the second point. The *payload variables* (e.g., `v1`, `b1`) allow the communicated *values* to be referred to below. Next, `P` has a choice between three cases that depends on the boolean values received from `R`. The `BothIn` case is given by the *refinement expression* (after the @) asserting `b1 and b2`. In this case, the `v2` payload specifies `P` to *forward* the second point to `C`—note: this value dependency is equivalent to specifying a fresh payload variable `x : Point` with `@(x=v2 and b1 and b2)`. The `BothOut` case informs `C` that neither point is above.

In the `Intrsct` case, `P` uses `R` to get the intersection of the current edge with the plane. `b1 xor b2` means only one point is above; and the payload values of `Intrsct` are the same points from earlier. The two subcases (line 18) determine whether or not the second point is forwarded with the intersection. Finally, the recursive `do Loop` (line 23) that follows all of the non-`Close` cases proceeds to the next iteration.

```
1 type SH_P = SessionTypeProvider<"SH.scr", "SH", "P", "config.yaml">
2             // Generates protocol- (SH) and role- (P) specific types
3 let P = new SH_P()
4 let main argv =
5   let s = P.Init() in // Start of P endpoint implementation
6   s .
      Send   State2 State1.Send(R Role, Plane label, Point x1, Point x2, Point x3, Point x4)
```

Figure 3. Instantiating the STP in F# for the P role in SH.

Session type providers: bridging MPST and F#. On the right of Fig. 1 are user programs. The flow of Fig. 1 is depicted in terms of inputs and outputs of the tool stages, but the actual workflow of the STP is driven on-demand in the reverse direction as the programmer uses the STP.

Fig. 3 gives a typical preamble when using the STP to implement a protocol endpoint. Line 1 instantiates the STP by supplying the Scribble module, the name of the root protocol and the target role as static parameters. This is the trigger for the F# compiler to engage the *compile*-time functionality of the STP, which we outline below with reference to the relevant sections with more detailed explanations.

(§ 3) First, the STP validates the specified protocol, according to the core MPST properties (such as reception errors and deadlock-freedom) and our extensions related to refinements. An invalid protocol causes a compile-time error to be raised here.

(§ 4) The STP performs the compile-time generation of the protocol- and role- specific types of an API for implementing this endpoint. The API presents a call-chaining interface: the target values represent the *state-specific* communication channels (*state channels*, for short), with methods for chaining I/O actions through states. More specifically:

- Each protocol state is generated as a distinct .NET class type, whose methods give exactly the I/O actions permitted at that state.
- The return type of each I/O method corresponds to the successor state of that action.

(§ 5) The compiler uses the STP for on-demand generation of the underlying API code, upon subsequent uses of the generated types in the user program. Besides the actual network I/O, the STP leverages the code generation to treat interaction refinements by a combination of correctness by construction and automated inlining of assertions.

As we shall illustrate below, the code generation depends on the kind of communication pattern induced by an interaction refinement, and the potential to enforce it *statically*.

The STP instantiation returns the frontend type of the API, named `SH_P` in Fig. 3. The instantiation on Line 3 represents creating a new session endpoint for this protocol and role, and line 5 starts its implementation. The `Init` method handles the connection actions for session initiation according to a local configuration file (`config.yaml`), and returns the *initial state channel*, bound to `s`.

```
1 let rec loop poly s =
2   match poly with
3   | v1::v2::tail →
4     let b1 = new Buf<bool>()
5     let b2 = new Buf<bool>()
6     let s' = s.Send(R, IsAbove, v1).Receive(R, Res, b1)
7               .Send(R, IsAbove, v2).Receive(R, Res, b2)
8     let s'' = // s'' has the same type as s
9       if b1.val then
10        if b2.val then s'.Send(C, BothIn).Send(R, BothIn) else isct s'
11      else
12        if b2.val then isct s' else s'.Send(C, BothOut).Send(R, BothOut)
13    loop (v2::tail) s'' // Recursion
14  | _ → s.Send(R, Close).Send(C, Close) // End
15
16 let isct p =
17   let s' = s.Send(R, Intrsct).Receive(R, Res, new Buf<Point>())
18   if b2.val then s'.Send(C, SecIn) else s'.Send(C, SecOut)
```

Figure 4. The main loop for P using the STP-generated API.

Distributed programming in F# using the STP. Starting from the initial state channel, an STP-generated API should be used according to one simple condition: *invoke* one *I/O method on the current state channel to obtain the next, continuing up to the end of the protocol* (if any).

The API generation includes utility constants for the various names in the protocol specification (e.g., roles and labels) following a *singleton type* pattern; we shall assume convenience `let`-bindings, e.g., `let R = SH_P.Role.R.val`, where `SH_P.Role.R` is the generated type for role `R` and `val` is the single value of this type. We can continue the implementation of `P` by changing line 6 in Fig. 3 to

```
loop poly (s.Send(R, Plane, p1, p2, p3, p4))
```

assuming `p1`–`p4` of type `Point` and `poly` of type `Point list`. The `loop` function, given in Fig. 4, corresponds to the `rec Loop` in the Scribble protocol. We summarise the key features of the STP-generated API, as demonstrated in the code of `loop`.

Static type checking Fig. 3 illustrates a standard IDE feature (here, Visual Studio IntelliSense) for listing the valid members of a type, as applied to the initial state channel `s`. As per the protocol, the only I/O action permitted there for `P` is to send `R` the `Plane` message. Similarly, the successor state channel offers

```
Send: R → IsAbove → Point → SH_P.State2
```

as used on line 6 in Fig. 4. Typing will thus statically detect protocol violations such as sending to the wrong role (i.e., `C`), sending an unexpected message (e.g., `Intrsct`) or payload (e.g., `true`), or attempting to `Receive` instead of `Send`.

Correctness by construction The protocol specifies `BothIn(v2) from P to C`, but line 10 in Fig. 4 shows the generated method is just `s'.Send(C, BothIn)`—i.e., no `v2` argument. Since `P` must forward to `C` exactly the `v2` value given earlier (in `IsAbove`), the STP generates the API at `P` to internally cache this value and implicitly send it in the `Intrsct` message, ensuring correctness by construction. More general refinement expressions (i.e.,

beyond simple forwarding) can be treated in the same manner, as explained in § 4.1.

Safe optimisations The interaction `Intrsct(v1, v2) from P to R` corresponds to `p.Send(R, Intrsct)` on line 17 in Fig. 4. Similarly to the above, this refinement states `P` must send `Intrsct` with the same `v1` and `v2` values given earlier. In this instance, however, `R` also already knows these values from the earlier interactions. The STP supports cross-role interaction optimisations, by generating, e.g., the API at `R` to internally cache and reuse those values—and the API at `P` to omit them from the `Intrsct` message.

Automated assertion inlining Combining types and code generation allows useful and common refinement patterns, like those above, to be statically realised in basic F#. As the default base case, a refinement expression (or possibly just some part of it) may be treated by directly inlining it into the generated code as a run-time assertion. For instance, there is no static assurance that the code for the `BothIn`/`Out`/`Intrsct` choice on lines 8–12 adheres to the specified refinements. Instead, the STP embeds, e.g., the assertion `b1 && b2` into the `Send(C, BothIn)` method body.

2.2 Safety Guarantees of STP-generated APIs

As stated earlier, the programming interface presented by the STP is based on a *linear* (exactly once) usage discipline on state channels. The STP enforces linearity by embedding run-time checks into the generated I/O methods: a simple boolean guard on whether the channel has already been "used" [14, 19]. This is similar to the base case assertion inlining—for both linearity checks and refinement assertions, the generated code handles a violation by raising an exception *without* performing the actual I/O action.

The STP thus offers the following safety guarantee for endpoint programs:

A statically *well-typed STP endpoint program will* never *perform a non-compliant I/O action w.r.t. the source protocol.*

This is because the *only* way a well-typed program can *attempt* a non-compliant I/O action is by violating linearity or an inlined assertion. If an endpoint program is well-typed but not fully correct, execution will at worst result in an unfinished protocol, which is always a caveat in practice due to program errors outside the protocol code or failures.

3 Specification and Validation of MPST with Interaction Refinements

We present our extensions to the Scribble protocol description language for specifying communication patterns with interaction refinements, and our implementation of protocol validation.

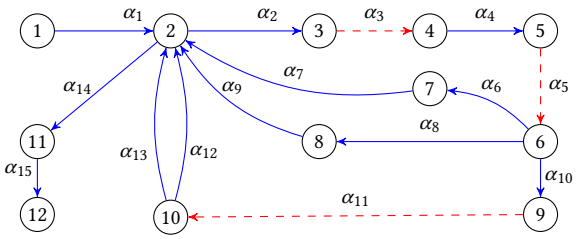
Syntax. Below is the syntax of global protocols P in our extended Scribble. We use the following base notations: p stands for protocol names; r stands for role names (concrete role names are `A`, `B`, etc.); l for message labels; S for data types (sorts, e.g., `int`, `bool`); x for variables; f for function names; and n for integers.

$$
\begin{array}{lcl}
P & ::= & \texttt{global protocol}\ p\ (\texttt{role}\ r_1, \ldots, \texttt{role}\ r_n)\ \{G\} \\
G & ::= & l(T)\ \texttt{from}\ r_1\ \texttt{to}\ r_2;\ @E \quad | \quad \texttt{do}\ p(r_1, \ldots, r_n); \\
 & | & \texttt{choice at}\ r\ \{G_1\}\ \texttt{or} \ldots \texttt{or}\ \{G_n\} \quad | \quad G\ G \\
T & ::= & x_1 : \mathrm{S}_1, \ldots, x_n : \mathrm{S}_n \\
E & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid E \oplus E \mid \ominus E \mid f(E_1, \ldots, E_n) \\
\oplus & ::= & \texttt{and} \mid \texttt{or} \mid = \mid < \mid > \mid + \mid * \qquad \ominus ::= \texttt{not} \mid -
\end{array}
$$

An *interaction* (from plain Scribble), $l(T)$ `from` r_1 `to` r_2, specifies that role r_1 *sends* asynchronously (i.e., without blocking) an l message with a payload T (a tuple of sorts S) to r_2, who blocks until the message is received. Our extensions allow (1) payload elements to be annotated by a variable x; and (2) the interaction to be refined by a boolean *refinement expression* $@E$, which specifies that the roles may perform their respective I/O actions only if E holds. E includes logical connectives, comparisons on arithmetic expressions and functions, on payload variables and constants. An interaction from plain Scribble is the same as a refined interaction with `@true`.

The other constructs are as in plain Scribble. `choice at` r specifies a branch point in the protocol where the *choice subject*, role r, decides which one of the `or`-separated cases the protocol should follow. The decision is made as an *internal* choice by (the implementation of) r, but must be explicitly communicated as an *external* choice to the other roles involved in each case. The `do` statement allows protocols to be composed from subprotocols and recursive protocol definitions (e.g., Fig. 2). Constructs are sequenced by $G\ G$. We shall omit the details of basic syntactic consistency checks that are as the reader would expect: e.g., to rule out occurrences of r (resp. x) not bound by `role` r (resp. payload element $x : \mathrm{S}$), unreachable code after a recursive `do`, and badly-formed expressions such as `3 < true`.

Protocol validation. The validation method of Scribble is based on a combination of (a) syntactic constraints, derived from the characteristics of formal MPSTs, with (b) explicit checking of MPST errors (e.g., reception errors and deadlocks) on an asynchronous model of the protocol [15]. The main idea is that (a) delimits a class of protocols for which it is *sound* to perform (b) on a finite model of the protocol with bounded-capacity channels: if the bounded model is error-free, then the protocol is error-free in general. In particular, Scribble builds on theoretical studies of MPST as communicating automata [7] where safety properties can be established from 1-bounded channels. Here, we omit the details of the syntactic constraints and base properties checked by Scribble, and focus on properties directly related to refinements. An overview of the omitted properties can be found at [26].



α_1 = R!Plane(p,p,p,p)
α_2 = R!IsAbove(v1:p)
α_3 = R?Res(b1:b)
α_4 = R!IsAbove(v2:p)
α_5 = R?Res(b2:b)
α_6 = C!BothIn(x_1); $x_1 \mapsto$ v2; b1∧b2
α_7 = R!BothIn()
α_8 = C!BothOut(); ¬(b1∧b2)
α_9 = R!BothOut()
α_{10} = R!Intrsct(); b1⊕b2
α_{11} = R?Res(i:P)
α_{12} = C!SecOut(x_2); $x_2 \mapsto$ i
α_{13} = C!SecIn(x_3,x_4); $x_4 \mapsto$ i, $x_4 \mapsto$ v2
α_{14} = R!Close()
α_{15} = C!Close()

Figure 5. CFSM representation of role P from protocol SH in Fig. 2 (→ denotes an output, ⇢ is an input)

Properties checked on refinements. To sum up the core notion of model employed in Scribble: states record the current position of each role in the protocol and the contents of its input queues from each peer, and transitions are the asynchronous send and receive actions by roles. We extend model states with two elements, K and F, to treat refinement expressions. K is a map from roles r to sets of variables $\{x_i\}_{i \in I}$ that records the *local* knowledge of payload variables at each role: sending or receiving an annotated payload element adds that variable to the K of the subject role. F is a map from roles r to expressions E that records the conjunction of refinement expressions passed by each role in its interactions so far.

Our implementation checks the following based on this extended model, supported by the Z3 SMT solver. Let S be any model state, with associated K and F, and let r be any role at an output point in the protocol in S.

Variable knowledge For all actions of r, r must know every variable occurring in the E of the corresponding interaction, i.e., $\forall x \in \text{vars}(E).x \in K(r)$.

```
1(x : int) from A to C;        2(y : int) from B to C; @y > x
```

The above is invalid as B does not know x (the wavy-underlined red font indicates the error in the Scribble code).

Refinement satisfiability For all actions of r, the E must be satisfiable for *some* interpretation of $F(r)$.

```
1(x : int) from A to B;                @x > 3
choice at B { 2() from B to A;         @true }
          or { 3(y : int) from B to A; @y>x+1 and y<4 }
```

The 2 case is always eligible, but the 3 case is never eligible. Such errors correspond to unreachable protocol flows (or protocol deadcode). Written in the SMT 2 language of Z3, our implementation catches this by asserting for B:

```
(and (> y (+ x 1))(< y 4)(> x 3))
```

Refinement progress r must have an action for which the E is satisfiable for *all* interpretations of $F(r)$.

```
1(x : int) from A to B; @x > 3        2(y : int) from C to B;
choice at B { 3() from B to A; @x<y }
          or { 4() from B to A; @x>y }
```

Above, B (and consequently A) would be stuck at the choice in the situation that $x = y$; this is caught by asserting for B:

```
(forall ((x Int)(y Int))(=> (> x 3)(or (< x y)(> x y))))
```

A fix would be to add the equality case to the choice. Adding @y<=3 to the 2 interaction would fix refinement progress but violate satisfiability. At the endpoint programming level, refinement progress ensures at any output point in the protocol implementation that there will *always* be some action for which E necessarily holds.

4 From Session Types to F# Types

After validating the source protocol (§ 3), the STP generates F# *types* (more precisely, .NET types) to capture the *structure* of the protocol from the perspective of the target role. This section explains the type generation, which is based on a representation of the localised protocol as a communicating finite state machine (FSM) with extensions for treating interaction refinements in the later *code* generation (§ 5).

4.1 From Scribble to CFSMs

Continuing our running example, Fig. 5 illustrates the I/O structure for the P role in the SH protocol (Fig. 2). We have shortened the types Point to p and bool to b. The initial state is 1. The notation, e.g., R!Plane(p,..,p) denotes the local *send* action by p to R of the specified message; ? denotes a local *receive* action. The outermost choice at P in Fig. 2 (line 6) corresponds to state 2, the first nested choice (line 9) to state 6, and the innermost choice (line 18) to state 10. The recursive protocol definition manifests as the cyclic paths back to state 2. Note that the CFSM follows the grammar in § 3 with regards to de-sugaring of message-value forwarding patterns: e.g., the v2 payload of BothIn in Fig. 2 becomes a fresh variable x_1 in α_6 (with an additional constraint, explained below).

Our extended CFSMs are defined as follows. An endpoint FSM is a tuple $(\mathbb{S}, \mathbb{R}, s_0, \mathbb{L}, \mathbb{T}, \delta)$, where $\mathbb{S} = \{s, s_1, s_2, ..\}$ is the set of *states*, $\mathbb{R}$ is the set of role names, s_0 is the *initial state*, $\mathbb{L}$ is the set of message labels, and $\mathbb{T}$ is the set of payload types. $\delta : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$ is the *transition function*, where $\mathbb{A} = \{\alpha_1, \alpha_2, ..\}$ is the set of *local actions*. Each α has the form $r \dagger l(x_i : T_i)_{i \in I}; \sigma; A$ with $r \in \mathbb{R}$, $\dagger \in \{!, ?\}$, $l \in \mathbb{L}$, and $\mathsf{S} \in \mathbb{T}$ for each S in T. We annotate an α by its kind, i.e., $\alpha^!$ and $\alpha^?$, and similarly $\sigma^!$ and $\sigma^?$; and define $\delta(s) = \{\alpha \mid \exists s'.\delta(s, \alpha) = s'\}$.

The σ and A elements are the main extensions: $\sigma^!$ and $\sigma^?$ are both maps from variables x to expressions E (but with different purposes), and A is an expression E. (In the above example, we omitted empty σ from actions; similarly when A

is `true`.) We now explain these elements using the example. We have highlighted in Fig. 5 and Fig. 4 where $\sigma^!$ (orange) and $\sigma^?$ (blue) manifest in the CFSM and user program.

Constructive variables $\sigma^!$ records expressions for payload variables to be *sent* where the expression can be *statically* determined from equality clauses in the refinement. These expressions are used to enforce the refinement *by construction* via the later code generation. E.g., in α_6, $\sigma = \{x_1 \mapsto \texttt{v2}\}$ is determined from the message-value forwarding pattern `BothIn(v2) from P`. Our implementation determines these expressions by checking the satisfiability of the implication from the current constraints (i.e., F) to the syntactically extracted candidate expressions (e.g., $x_1 = \texttt{v2}$, for which this check trivially holds since x_1 is fresh).

Interaction optimisations $\sigma^?$ records expressions for payload variables that do *not* need to be sent because the receiver already has the values needed to compute the expression locally. Our implementation determines these directly from variables occurrences in the protocol syntax. Such variables are then elided from T at the sender-side. E.g, in α_{10}, both `v1` and `v2` have been elided from the `Intrsct` message; in the corresponding receive action at `R`, $\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2\}$ where $x_{1,2}$ are fresh variables from de-sugaring the source Scribble.

Assertion A is derived from the source refinement expression, taking into account the constructive variables. In α_6, $A = \texttt{b1} \wedge \texttt{b2}$—this is the result of pruning the x_1 clause from the underlying expression $x_1 = \texttt{v2} \wedge \texttt{b1} \wedge \texttt{b2}$ given by de-sugaring the source syntax. Refinements that are statically determined to be `true` are simply pruned (useful for E as *design*-level assertions).

Value forwarding patterns (like α_6) are a special case of constructive variables. The principle applies to any expressions that can be statically determined in the above manner.

4.2 From CFSMs to F# Types

The STP uses the CFSM to generate protocol- and role- specific types for the endpoint API. All of the generated types are housed as nested types in the frontend type returned by instantiating the STP (e.g., `SH_P` in Fig. 3). First, the STP generates utility constants for each member of $\mathbb{R}$ and $\mathbb{L}$ by a singleton type pattern: e.g., for role `P`, the STP generates a nested class `SH_P.P` and a constant `SH_P.P.val` that is the sole value of this type. These are used to direct the I/O operations described below. The frontend class has a method `Init()` that uses the `config.yaml` supplied to the STP to perform the local connection operations for initiating a new session.

Second, the STP generates a family of class types that captures the I/O structure of the FSM for this endpoint. This relies on the following properties of FSMs derived from a *valid* source protocol [14]: (1) all actions at each state are of the same kind, i.e., a state cannot have both ! and ? actions; and (2) ?-actions at any given state specify the same r. The generation uses a map $[\![s]\!]$ from states s to distinct .NET class names. The user may provide meaningful names for states, or else the STP generates default names by enumerating the states. Each state s is generated as a class type as described by the following cases:

Output $\delta(s) = \{\alpha_i^!\}_{1..n}, n > 0$: the STP generates
`type` $[\![s]\!]$ = $m_1 \ldots m_n$
where for each $\alpha_i = r_i!l_i(T_i); \sigma_i; A_i$,
m_i = `member Send:` $r_i \rightarrow l_i \rightarrow \phi_!(T_i, \sigma_i) \rightarrow [\![\delta(s, \alpha_i)]\!]$

Single-input $\delta(s) = \{r?l(T); \sigma; A\}$: the STP generates
`type` $[\![s]\!]$ = `member Receive:` $r \rightarrow l \rightarrow \phi_?(T) \rightarrow [\![\delta(s, \alpha)]\!]$

Branch-input $\delta(s) = \{\alpha_i^?\}_{i \in I}, |I| > 1$: the STP generates
`type` $[\![s]\!]$ = `member Branch:` $r \rightarrow I_{[\![s]\!]}$
`interface` $I_{[\![s]\!]}$
and for each $\alpha_i = r?l_i(T_i); \sigma_i; A_i$, a nested type within $[\![s]\!]$:
`type` C_{l_i} = `interface` $I_{[\![s]\!]}$ `with`
`member Receive:` $\phi_?(T_i) \rightarrow [\![\delta(s, \alpha_i)]\!]$

In the above, $\phi_!((x_i : \mathsf{S}_i)_{1..n}, \sigma) = [\mathsf{S}_j]_{[j \mid j \in 1..n \wedge x_j \notin \sigma]}$ and $\phi_?((x_i : \mathsf{S}_i)_{1..n}) = [\mathsf{Buf}\langle \mathsf{S}_i \rangle]_{1..n}$, where the notation $[\mathsf{S}_i]_{1..n}$ stands for $\mathsf{S}_1 \rightarrow \ldots \rightarrow \mathsf{S}_n$.

For example, the STP generates for state 10 in the FSM of `P` (assuming default state naming):

```
type S10 = member send: C → SecOut → S2 member send: C → SecIn → S2
```

Both actions take no payload arguments due to their constructive variables, and lead to the same successor state.

For branch-inputs, the `branch` method will return a case-specific class type C_{l_i} that implements the intermediate interface type $I_{[\![s]\!]}$ for this branch. Each case type has *only* the `receive` for that case, that takes a `Buf` for each payload value and returns the successor state. The appropriate case should be safely determined by a standard F# type test pattern. For example, the outermost choice for `R` in Fig. 2 corresponds to a branch-input state s_2 where $\delta(s_2) = \{\texttt{P?isAbove(v1:P)}, \texttt{P?Close()}\}$ and $[\![s_2]\!] = \texttt{S2}$. This branch may be implemented using the STP-generated API by

```
match s.branch(P) with // By the type generation, only two cases
| :?  SH_R.S2.IsAbove as s1 →
  let p1 = new Buf<Point>()
  let p2 = new Buf<Point>()
  f (s1.Receive(p1).Send(P, Res, (isAbove (p1.val)))
       .Receive(P, p2).Send(P, Res, (isAbove (p2.val)))) p1 p2
| :?  SH_R.S2.Close as s2 → s2.Receive(P) // Protocol end
```

where `s` is of type `S2`, `SH_R` is the frontend type given by instantiating the STP for `SH` and `R`, `isAbove` is a local auxiliary function, and `f` is a function that finishes the `IsAbove` case.

5 STP Code Generation

We outline the implementation of the STP, and explain the code generation related to static and dynamic enforcement of refinement expressions. F# has several integrated features for meta programming that we use to support on-demand generation of types and code from Scribble protocols: (1) *quotations* for programmatically creating and representing code

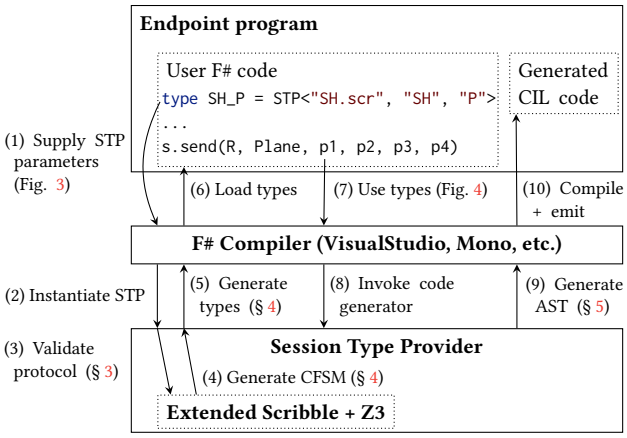


Figure 6. STP Workflow of types and code generation.

as values; (2) AST *splicing* for composing code fragments (expression trees and literals); and (3) the *type provider* framework itself, as a means for plug-in compiler extension.

5.1 Session Type Provider Implementation

Our session type provider is implemented as a *generative* type provider—this kind of type provider produces types that are added as concrete definitions in the final compiled program. The alternative are *erasing* type providers, for which the produced types are erased at run-time (e.g., the types of object values are erased to `obj`). Our current approach relies on generated types for statically safe branch handling via type test patterns (§ 4.2).

Compile-time operation of the STP. Fig. 6 depicts the workflow of the STP, driven on-demand during the *compile*-time of the user program. As explained through the preceding sections, the flow starts with the instantiation of the STP by the user in (1) in Fig. 6, leading to the loading of generated types by the compiler in (6).

Below is a snippet from the STP implementation for generating a `send` method from an $\alpha = r!l(T); \sigma; A$ in the CFSM.

```
let getBody ps = // ps are the method parameters
   let astVals = makeVA T, V, A, ps // makeVA explained in § 5.2
   <@ let payloadVals = %astVals ...
      Transport.send this.sessid %ps.[0] %ps.[1] payloadVals
   @>  // getBody returns an AST given as a quotation
outputC.addMethod("send", r, l, T, getBody) // r, l, T from CFSM α
```

`outputC` is a value that represents an output-state class under construction via the type providers API. The call to `addMethod` is performed in (5) according to the generation of class types and members for CFSM output states in § 4.2. The arguments to `addMethod`, except the last, specify the signature (name and parameter types) of the method being constructed—r, l and T correspond to the elements of α.

Subsequent uses of the generated types, e.g., the `send` invocation in step (7), triggers the generation of the actual code for the target operation. The last argument to `addMethod`, the function `getBody`, is called by the type provider framework in (8) to construct an abstract syntax tree (AST) for the method body. The AST is specified by the quotation (`<@..@>`), which is a template for an expression with parameters (prefixed by `%`) to be filled by the compiler. The compiler passes the parameter names of the method being constructed as a list of *expression* values as `ps`, and evaluates `getBody` to an AST by splicing the relevant expressions into the quotation at the `%`-positions. Finally, the AST for the generated code is compiled into the .NET Common Intermediate Language (CIL) and emitted to the endpoint program.

Networking code. The code inside the quotation shown above includes a call to the `send` operation of the networking library accompanying the STP. This operation takes as arguments (via the spliced in parameters) the values for the session ID, target role, the message label, and the payload values to be communicated.

We briefly outline the implementation of the networking library. The generated `Init` method of the API frontend type (§ 4.2) obtains from the local `config.yaml` the transport kind (by default, TCP), and a remote network address or local port to connect/accept on for each session peer. The `Init` method returns when a connection has been established to each peer (or throws an exception on any error). The networking library at each endpoint maintains a mapping from session instances and peer roles to the concrete network connections internally, and dispatches messages on connections according to the channel instance and role argument (e.g., `s` and `R`) supplied by the user to the generated code.

5.2 Static and Dynamic Treatment of Refinements

Code generation for constructive variables. So far, we have explained the STP code generation related to the core I/O structure of an endpoint FSM, i.e., the protocol states s and the role r, label l and payload T elements of actions α. The aspects of code generation related to enforcing the specified refinements stem from the statically analysed σ and A elements of α (§ 4.1). Below is a pseudocode outline of the *generated code* (TAST) returned by the `makeVA` function used in the earlier snippet, which treats σ and A at the sender-side.

```
makeVA(T, σ, A, ps) =          // where T = (x_i : S_i)_{1..n}; let j := 2
<@ for i in 1..n
      if x_i ∈ dom(σ) then
         (σ(x_i), C) ⇓ v_i; C[x_i ↦ v_i]     (1a) Evaluate var expr
      else
         C[x_i ↦ %ps.[j]]; j++               (1b) Take next ps
if(A, C) ⇓ false then exception               (2) Check assertion
return C(x_1), ..., C(x_n) @>                 (3) Return payloads
```

Let *cache* C be a map from variables x to values v. We write $C[x \mapsto v]$ to stand for the operation of updating the value of x in C to v (leaving other variables unchanged);

and $(E, C) \Downarrow v$ for the evaluation of an expression E under the context of C to v. ps is the list of parameter expressions described in § 5.1; the index j starts at 2 for the payload parameters (i.e., skipping the role and label).

For each element of T, the generated code will update cache C either (1a) by evaluating the expression E given by σ in the case of constructive variables, or (1b) directly from the user-supplied ps. It then (2) checks the assertion, using the latest values of any constructive variables; and (3) and returns the concrete payload values to be sent, in order.

The code generation allocates a cache per instance of the STP frontend type (i.e., per endpoint—e.g., `new SH_P` in Fig. 3). Protocol validation (§ 3) guarantees (a) the latest value of a variable is present in C when used in any E_i or A; and (b) in conjunction with run-time checks on linear channel usage (explained below) precludes race conditions on variable access. For simplicity, we have portrayed C as recording all refinement expression variables; our implementation caches only those that *may* be needed to compute a constructive variable or check an A, determined straightforwardly by Scribble from the syntax of the source protocol.

Assertion inlining. For the $(E, C) \Downarrow$ operations, the code generation wraps E in a lambda expression with the free variables of E (ordered lexicographically) as parameters. This lambda expression is applied to the *run-time* values of each variable in C, i.e., $(\texttt{fun}\ x_1 \ldots x_n \rightarrow\ E)\ C(x_1) \ldots C(x_n)$ where $x_1 \ldots x_n$ are the free variables of E. Thus, for interaction refinements that are not fully statically covered by the previous code generation steps, the (remaining) assertion A is dynamically enforced by evaluating it as a boolean predicate. Our implementation allows the user to disable assertion inlining for `Receive` methods. This is a safe optimisation if the user trusts in the correctness of the other session endpoints: if all endpoints are STP-implementations (or are otherwise correct), then sender-only assertion inlining is sufficient to guarantee the STP safety properties.

As explained in § 2.2, every generated I/O method also has an inlined run-time check for linear channel usage. Each channel instance simply has a boolean "used" flag that the generated code will check on method entry: an exception is thrown if the channel has already been used (without performing the offending I/O action), or else the flag is set.

6 Evaluation

This section evaluates compile-time and run-time performance. Although we have not as yet considered performance as a primary consideration in our current implementation, these preliminary results demonstrate the applicability of our approach.

Setup. We use Visual Studio 2015, .NET Framework 4.5.2 and F# runtime 4.4.0.0. Our machine configurations are Windows 7 Enterprise 64-bit; Intel Core i7-3770 @ 3.40GHz; 16Gb RAM. We repeated each benchmark 30 times and report the average.

Table 1. Time taken by STP types generation.

Example (role)	#LoC	#States	#Types	Gen (ms)
2-Buyer (B_1) [13]	16	7	7	280
3-Buyer (B_1) [5]	16	7	7	310
Fibonacci (S) [14]	17	5	7	300
Travel Agency (A) [24]	26	6	10	278
SMTP (C) [14]	165	18	29	902
HTTP (S) [3]	140	6	21	750
SAP-Negotiation (C) [18]	40	5	9	347
Supplier Info (Q) [24]	86	5	25	1582
SH (P)	30	12	15	440

For run-time performance, we ran each process on a separate machine with latency measured to be 0.24 ms (ping 64 bytes).

Compile-time performance of the STP. We use examples from three categories: (1) classical examples from session types literature, to confirm the STP supports core session types features; (2) real-world use cases, including subsets of SMTP and HTTP that demonstrate interoperability between STP-implemented programs and existing client/servers (e.g., Microsoft Exchange, Firefox and Apache); and (3) micro-benchmarks to stress test scalability. We measure the time taken by the STP to generate all types and AST values from the CFSM (from the start of step 5 to the end of 9 in Fig. 6).

Table 1 reports the results for (1) and (2). LoC refers to Scribble protocol; states to the *local* CFSM for the target role. We extended some of the original session types with expected uses of refinements: e.g., in Fibonacci, to express the addition as a value constraint; in 2-Buyer, to express that the same quote value is sent to both buyers, and the split in cost between them. In HTTP, the size of a message body is determined by the *value* of the `Content-Length` header field,

```
CONTLEN(n:int)from C to S; ... BODY(b:string)from C to S; @size(b)=n
```

where `size` is a built-in function of our extended Scribble. Travel Agency is from a W3C Choreographies use case; SAP-Negotiation is for Service Agreement Proposals; and Supplier Info is based on a microservices use case.

For (3), we use protocols of increasing size (number of states and generated types), formed by repeated composition of the following pattern, where i ranges over $1..n$:

$\texttt{ping}_i\texttt{(int) from A to B; @}E\quad \texttt{pong}_i\texttt{(int) from B to A}$

(The specific protocol structure is less important here as the STP conducts a single linear pass over the state set; § 4.2.) Fig. 7 (a) reports the results for n from 10 to 50, where **P1** has an empty E, and **P2** has $E = \texttt{x}_i > \texttt{x}_{i-1}$. The `Scr` part of the result is the time from STP instantiation to CFSM generation, and `Gen` the time from there to finish types/AST generation.

Overall results. The generation time is less than two seconds in all cases of (1) and (2): it had negligible effect on the programming experience (note for stable protocols, generation is a one-time compilation cost). (3) confirms the linear cost of generation in the number of local endpoint states.

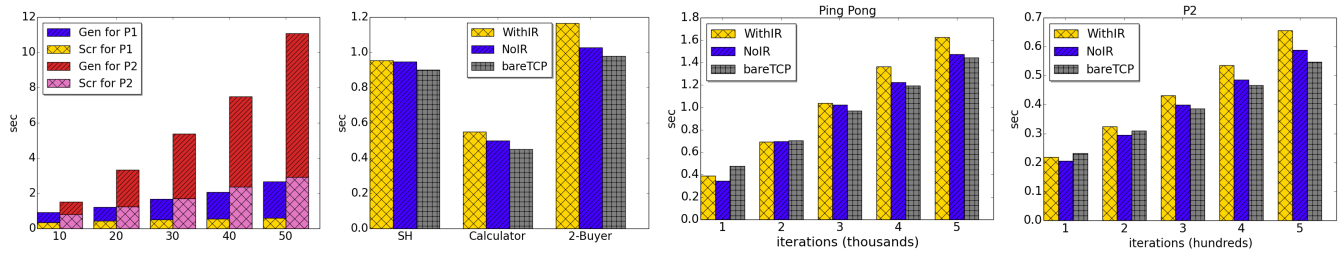


Figure 7. From left to right: **(a)** compile-time performance for **P1** and **P2**; and run-time performance for **(b)** SH, Calculator and 2-Buyer; **(c)** Ping-Pong; and **(d) P2**.

Run-time performance of STP-implemented programs. We conducted several benchmarks to evaluate how the design of the STP impacts the run-time for applications using TCP sockets. Any overheads introduced by the STP stem from: (i) *type instantiation and use*—e.g., state channel creation and branch typecases; (ii) *linearity checks*—checking and setting the boolean flag on state channel use; and (iii) *inlined assertions*—run-time enforcement of refinements. For each application, we compare three versions: an STP implementation with interaction refinements (WithIR); an STP implementation without refinements (NoIR); and an "untyped" implementation that directly uses the standard .NET TCP library (System.Net.Sockets) and enums for pattern matching messages in branches (bareTCP). We measure the time after session initiation to end.

Fig. 7 (b–d) report the results for the following applications, all implemented in F#. We explain each and its interaction refinement(s) (IR for short), and summarise the results. We say TCP-overhead for NoIR against bareTCP; and IR-overhead for WithIR against NoIR.

SH is our running example on a polygon of 100 points. TCP-overhead is 5%, while the IR-overhead is only 1%—as a more realistic application, most of the time was taken by the calculations at R.

Calculator is a distributed service for arithmetic operations ($+$, $-$, $\times$ and $\div$). The recursive protocol allows a client to repeatedly send an operation request with two numbers x and y, and receive back the result. IR are accordingly specified for each operation; the case of division also specifies $y \neq 0$. The results are for a client requesting 100 operations. TCP-overhead is 9%. For IR-overhead, we the constructive variables to check *all* refinements dynamically as an artificial worst case, resulting in 11% overhead.

2-Buyer is a negotiation between two buyers B_1 and B_2, and a seller S. B_1 names items for purchase to S, who sends a quote to both B_1 and B_2; B_1 tells B_2 how much she will contribute, and B_2 notifies S whether or not she accepts. An IR at S specifies the same quote is sent to both buyers, and an IR at B_2 specifies to accept if the contribution offered by B_1 is more than half. The results are for 1000 interactions. TCP-overhead is 4.8%, and IR-overhead is 13%.

Ping-Pong is a micro-benchmark given by wrapping the `ping-pong` pattern from earlier in a recursion, with E =`x`> 0 and the second payload changed to `x` (value dependency), that is repeated the specified number of times. (Unlike **P1** and **P2**, this protocol is a fixed small size.) Similarly to above, TCP-overhead stays below 5%, while IR-overhead is 12%.

P2 is as defined earlier with n=100, which is then repeated in a recursion 100–500 times. In contrast to **Ping-Pong**, this protocol generates a large number of types and different assertion objects. However, IR-overhead remains similar (below 11%), indicating that the number of assertion objects does not significantly affect performance; while TCP-overhead is slightly higher at 5-7%, due to the increased type loading.

Summary. Our evaluation shows that the compile-time cost of the STP scales w.r.t. the protocol size, with low run-time overhead compared to the directly implementing the TCP base cases from scratch, i.e., without any assistance from specification-derived types or code generation. The primary motivation of our work is for such programmatic and safety benefits. We found using the STP also reduces bugs and code size by simplifying several error prone aspects, such as socket creation and message serialization; and that the types are valuable for exploratory programming and documentation, particularly so compared to large prose-based specifications such as the HTTP and SMTP RFCs. The automated treatment of refinements by the STP also allows the programmer to add/remove checks without modifying the endpoint code. Our run-time results are from a low latency environment; in practice, network latency will further offset any overheads.

7 Related Work

We summarise the most closely related works on extensions of session types, and code generation for protocol APIs. See [26] for further discussions.

Theoretical works on session types with refinements. The earliest work [4] (and closest to our extended Scribble) extends the core MPST theory (session π-calculus and type system [13]) to a Design-by-Contract methodology based on projecting global assertions. The variable knowledge and refinement progress properties (§ 3) that we check

are similar to their syntactic well-formedness of global assertions, but developed for our practical setting via model checking and SMT solving—which permits more expressive interaction structures than the core MPST theory (e.g., [4] restricts any third party in a choice to the same behaviour in all cases), and will facilitate integration with other practical MPST extensions (e.g., [15]). [2] presents a binary (two-party) session π-calculus where assumptions and assertions are modelled as explicit linear resources: assertion predicate terms are cut (i.e., consumed) against (syntactically) matching assumption context terms; [8] is a direct implementation as an executable π-calculus with syntactic predicate matching. [29] extends an MPST framework with value-dependent types from [28]. [29] is based on explicitly exchanging proof objects, whose validity is checked at run-time, to confirm the consistency of data constraints at the two ends. In contrast to these works, the focus of this paper is a practical development of MPST with interaction refinements for a real-world language (F#, and .NET), through a combination of static typing, code generation and tools integration.

Session programming in mainstream languages. Several recent works have studied applications of session types in statically-typed mainstream languages. This paper builds on the "hybrid" approach of [14] (Java)—code generation for static typing backed up by automated inlining of run-time linearity checks—to support and enforce communication patterns involving interaction refinements in native F#. Other works that use run-time linearity checking are [23] (Scala) and [19] (OCaml), based on a continuation-passing style for binary sessions; and [22], which implements Scala API generation for a subset of Scribble corresponding to a core syntactic formulation of MPST theory. Another approach for binary sessions is via monadic embeddings into the target type system, as in [21] (Haskell) and [16] (OCaml), which rely on relatively advanced typing facilities; error messages may be indirect or obfuscated by the embedding. None of these works support multiparty sessions with interaction refinements (all are for binary sessions or the core MPST only).

To the best of our knowledge, this paper is the first to apply session types using type providers, exploiting language-integrated compile-time generation of typed APIs to make session types practically accessible to programmers.

Acknowledgments

We thank the referees for their comments. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1,EP/N027833/1 and EP/N028201/1. The first author is supported by an EPSRC Doctoral Prize Fellowship.

References

[1] D. Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
[2] P. Baltazar, D. Mostrous, and V. T. Vasconcelos. Linearly refined session types. In *LINEARITY*, volume 101 of *EPTCS*, pages 38–49, 2012.
[3] *Behavioural Types: from Theory to Tools.* River Publishers, 2017.
[4] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
[5] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 26(2):238–302, 2016.
[6] D. Delimarsky. WsdlService Type Provider (F#). https://msdn.microsoft.com/visualfsharpdocs/conceptual/wsdlservice-type-provider-%5bfsharp%5d.
[7] P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
[8] J. Franco and V. T. Vasconcelos. A concurrent programming language with refined session types. In *BEAT*, volume 8368 of *LNCS*, pages 33–42, 2013.
[9] T. S. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.
[10] F# data: Library for data access. http://fsharp.github.io/FSharp.Data/.
[11] S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, Cambridge University Press, December 2009.
[12] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
[13] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.
[14] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016.
[15] R. Hu and N. Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *LNCS*, pages 116–133, 2017.
[16] K. Imai, N. Yoshida, and S. Yuen. Session-ocaml: A session-based library with polarities and lenses. In *COORDINATION*, volume 10319 of *LNCS*, pages 99–118. Springer, 2017.
[17] B. Nordström and K. Petersson. Types and specifications. In *IFIP Congress*, pages 915–920, 1983.
[18] Ocean Observatories Initiative. http://www.oceanobservatories.org/.
[19] L. Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.
[20] T. Petricek, G. Guerra, and D. Syme. Types from data: making structured data first-class citizens in F#. In *PIDL*, pages 477–490, 2016.
[21] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Haskell'08*, pages 25–36, New York, NY, USA, 2008. ACM.
[22] A. Scalas, O. Dardha, R. Hu, and N. Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31, 2017.
[23] A. Scalas and N. Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28, 2016.
[24] Scribble home page. http://www.scribble.org.
[25] O. Shivers and M. Might. Continuations and transducer composition. In *PLDI*, pages 295–307. ACM, 2006.
[26] Project page. https://session-type-provider.github.io/.
[27] D. Syme et al. F#3.0: Strongly-typed language support for internet-scale information sources.
[28] B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP*, pages 161–172. ACM, 2011.
[29] B. Toninho and N. Yoshida. Certifying data in multiparty session types. *J. Log. Algebr. Meth. Program.*, 90:61–83, 2017.
[30] Z3 SMT solver. http://z3.codeplex.com/.