Fluent Session Programming in C#

Shunsuke Kimura Keigo Imai
Gifu University, Japan
kimura@ct.info.gifu-u.ac.jp keigoi@gifu-u.ac.jp

We propose `SessionC#`, a lightweight session typed library for safe concurrent/distributed programming. The key features are (1) the improved fluent interface which enables writing communication in chained method calls, by exploiting C#'s out variables, and (2) amalgamation of session delegation with async/await, which materialises session cancellation in a limited form, which we call *session intervention*. We show the effectiveness of our proposal via a Bitcoin miner application.

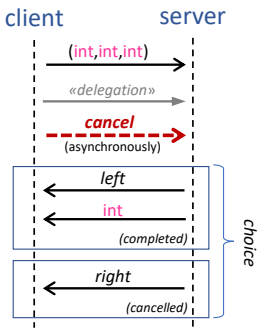
1 Introduction

Session types [8] are a theoretical framework for statically specifying and verifying communication protocols in concurrent and distributed programs. Session types guarantee that a well-typed program follows a *safe* communication protocol free from reception errors (unexpected messages) and deadlocks.

The major gaps between session types and "mainstream" programming language type system are the absence of the two key features: (1) *duality* for checking the communication protocol realises reciprocal communication actions between two peers, and (2) *linearity* ensuring that each peer is exactly following the protocol, in the way that channel variables are exclusively used from one site for the exact number of times. Various challenges have been made for incorporating them into general-purpose languages including Java [12], Scala [28], Haskell [25, 14, 19, 23], OCaml [24, 13] and Rust [15, 16].

We observe that the above-mentioned gaps in session-based programming can be *narrowed* further by the recent advancement of programming languages, which is driven by various real-world programming issues. In particular, C# [1] is widely used in areas ranging from Windows and web application platforms to gaming (e.g. Unity), and known to be eagerly adopting various language features including async/await, reifiable generics, named parameters, `out` variables and extension methods.

In this paper, we propose **`SessionC#`** – a library implementation of session types on top of the rich set of features in C#, and show its usefulness in concurrent/distributed programming, aiming for *practicality*. Namely, (1) it has an improved *fluent interface* (i.e., method calls can be chained) via C#'s `out` variables, reducing the risk of linearity violation in an *idiomatic* way. Furthermore, (2) it enables *session cancellation* in a limited form — which we call *session intervention* — by utilising amalgamation of C#'s async/await and *session delegation* in thread-based concurrency.



We illustrate the essential bits of `SessionC#` where a *cancellable* computation is guided by session types, by a use-case where a C# thread calculates a cancellable `tak` function which is designed to have a long running time [20]. The figure on the right depicts the overall communication protocol, which can be written in `SessionC#` as a ***protocol specification*** describing a client-server communication protocol from the client's viewpoint, as follows:

```
var prot = Send(Val<(int,int,int)>, Deleg(chan:Recv(Unit,End),
                                          Offer(left:Recv(Val<int>, End), right:End)));
```

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 61–75, doi:10.4204/EPTCS.314.6

```
1  var cliCh = prot.ForkThread(srvCh => {
2   var srvCh2 =
3    srvCh.Receive(out int x, out int y, out int z)
4          .DelegRecv(out var cancelCh);
5   cancelCh.ReceiveAsync(out Task cancel).Close();
6   try
7   {
8    var result = Tak(x, y, z);  // compute tak
9    srvCh2.SelectLeft().Send(result).Close();
10  }
11  catch (OperationCanceledException)
12  {
13   srvCh2.SelectRight().Close(); // if cancelled
14  }
15  int Tak(int a, int b, int c) {
16   if (cancel.IsCompleted)
17    throw new OperationCanceledException();
18   return a <= b ? b :
19    Tak(Tak(a-1,b,c),Tak(b-1,c,a),Tak(c-1,a,b));
20  }});
```

Figure 1: A Cancellable `tak` [20] Implementation in `SessionC#`

From the above, C# compiler can *statically* derive both the client and the server's *session type* which is *dual* to each other, ensuring the safe interaction between the two peers. The client starts with an output (`Send`) of a triple of integer values (`Val<(int,int,int)>`) as arguments to `tak` function, which continues to a *session delegation* (`Deleg`) where a channel with an input capability (`Recv(Unit,End)`) is passed — annotated by a *named parameter* *`chan`* — from the client to the server so that the server can get notified of *cancellation*. `Offer` specifies the client offering a binary choice between *`left`* option with a reception (`Recv`) of the resulting `int` value, and *`right`* option with an immediate closing (`End`), in case `tak` is cancelled.

The `tak` server's ***endpoint implementation*** in Figure 1 enjoys *compliance* to the protocol `prot` above, by starting itself using `ForkThread` method of protocol `prot`. It runs an anonymous function communicating on a channel `srvCh` passed as an object with the exact communication API methods prescribed by the session type, which is derived from the protocol specification. Note that the channel `cliCh` returned by `ForkThread` has the session type enforcing the client as well (which will be shown by Figure 5 of § 2.2.4).

Notably, the use of ***improved fluent interface*** in Line 3-4 enhances protocol compliance, where the consecutive *input* actions (`Receive`) are realised as the *chained* method calls in a row, promoting *linear* use of the returned session channels. The `out` keywords in Line 3 are the key for this; they declare the three variables `x`, `y` and `z` *in-place*, and upon delivery of the integer triple, the received values are bound to these variables (as their references are passed to the callee). In Line 4, `DelegRecv` accepts a delegation from the client, binding it to `cancelCh`. The protocol for `cancelCh` is inferred via `var` keyword as `Recv(Unit,End)` specifying the reception of a cancellation. The continuation is then assigned to `srvCh2`.

In addition, our design of the fluent interface also takes advantage of the modern *programming environment* like Visual Studio, via ***code completion***: The code editor suggests the correct communication primitive to the programmer, guided by the session types. The screenshot in Figure 2 is such an example in Visual Studio where the two alternatives in a `Select` branch are suggested right after the symbol '.' (dot) is typed.

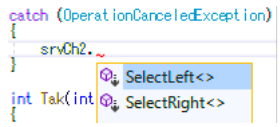


Figure 2: Completion

Furthermore, we claim that the ***session intervention pattern*** emerging from Line 5 is a novel intermix of C#'s async/await and session types in `SessionC#`, where a control flow of a session is affected by a delegated session. The delegated session can be seen as a *cancellation token* in folks, modelled by session-types. Line 5 schedules *asynchronous* reception of cancellation on `cancelCh` (`ReceiveAsync`), immediately returning from the method call (i.e., non-blocking) and binding the variable **`cancel`** to a `Task`. The task **`cancel`** gets *completed* when a `Unit` value of cancellation is delivered. The following `Close` leaves the reception incomplete. The task is checked inside `Tak` function (Line 16), raising `OperationCanceledException` if cancellation is delivered before finishing the calculation, which is caught by the outer `catch` block in Lines 11-14. Lines 8-9 in a `try` block calls the `Tak` function and sends (`Send`) the result back to the client

after selecting the left option (`SelectLeft`). If a cancellation is signalled by an exception, in `catch` block (Line 13) the server selects the right option (`SelectRight`) and closes the session.

See that all interactions in Figure 1 are deadlock-free, albeit binary session type systems like [8] and its successors used in many libraries, including ours, do not prevent deadlocks in general[1]. In the case above, operations on the delegated session `cancelCh` are non-blocking, and the session type on `srvCh` guarantees progress, provided that the client respects the *dual* session types as well, which is the case in the client in Figure 5 shown later (§ 2.2.4). Note that, however, in general, the program using *blocking* operations of `Task` may cause a deadlock.

Notes on Nondeterminism. The session intervention above is *nondeterministic*, as the client can send a cancellation *at any time* after it receives `cancelCh`. There is a possibility where the server may *disregard* the cancellation. For example, the server will *not* cancel the calculation if the client outputs on `cancelCh` *after* the server check `cancel` on Line 16, and after the result has been computed (i.e., no recursive call is made at Lines 18-19). In this case, the cancellation is still delivered to the server, and silently *ignored*. Note that the channel `cancelCh` is still used faithfully according to its protocol `Recv(Unit,End)` and *session fidelity* is maintained, as the reception is already "scheduled" by `ReceiveAsync` on Line 5. Note also that there is *no* confusion that the client consider that the calculation is cancelled, as the client must check the result of a cancellation via `Offer`, which we will revisit in § 2.2.4.

Notes on Linearity. C# does not have linear types, as in the most of mainstream languages. Thus there are two risks of linearity violations which are not checked statically: (1) use of a channel more than once and (2) channels discarded without using. For (1), we have implemented dynamic checking around it, which raises `LinearityViolationException` when a channel is used more than once. Regarding (2), although current `SessionC#` does not have capability to check it, we are planning to implement it around the *destructor* of a channel which is still not optimal, however better than nothing, as the check is delayed to the point when the garbage collector is invoked.

Notes on Session Cancellation. There are a few literature on session cancellation, such as Fowler et al.'s EGV [7], which we do not follow for now. Instead, The session intervention pattern above uses branching (`Select`/`Offer`) to handle a cancellation. There are a few issues on session cancellation in this form: (a) The cancellation handler clutters the protocol as the number of interaction increases, as mentioned in [7, § 1.2]. Although the branching-based solution is suitable for a short protocol like the above, there is a criticism by a reviewer specifically to `SessionC#` that (b) it lacks an exception handling mechanism, such as *crashing* (e.g. unhandled exceptions) and *disconnecting* (e.g. TCP connection failure). While we are yet to implement exception handling mechanisms, the distributed version of `SessionC#` equips `SessionCanceller` which handles session disconnection in terms of [7, 16].

Based on the key features and notes above, in the following sections, we explore the design space of modern session programming in `SessionC#`, showing the effectiveness of our proposal. The rest of this paper is structured as follows: In § 2, we describe the basic design of `SessionC#`, and show a few application in § 3. We conclude with remarks in § 4. Appendix § A describes implementation detail of the protocol combinators, and Appendix § B discusses more details on delegating a recursive session. Appendix § C includes more examples using `SessionC#`, including distributed implementation. The `SessionC#` is available at the following URL:

`https://github.com/curegit/session-csharp/`

[1] Exceptions are GV [30, 18] and its successors (e.g. EGV [7]), and Links language [5].

Session type	Synopsis	Combinator*	Return type*
`Send<V,S>`	Output `V` then do `S`	`Send(v,p)`$^{(1),(2)}$	`Dual<Send<V,S>,Recv<V,T>>`
`Recv<V,S>`	Input `V` then do `S`	`Recv(v,p)`$^{(1),(2)}$	`Dual<Recv<V,S>,Send<V,T>>`
`Select<SL,SR>`	Internal choice between `SL` and `SR`	`Select(left:`p_L`,right:`p_R`)`$^{(3)}$	`Dual<Select<`S_L`,`T_L`>,Offer<`S_R`,`T_R`>>`
`Offer<SL,SR>`	External choice between `SR` and `SR`	`Offer(left:`p_L`,right:`p_R`)`$^{(3)}$	`Dual<Offer<`S_L`,`T_L`>,Select<`S_R`,`T_R`>>`
`Eps`	End of the session	`End`	`Dual<Eps,Eps>`
`Goto0`	Jump to the beginning	`Goto0`	`Dual<Goto0,Goto0>`
`Deleg<S0,T0,S>`	Delegate `S0` then do `S` (where `T0` is dual of `S0`)	`Deleg(chan:`p_0`,p)`$^{(2),(3)}$	`Dual<Deleg<`S_0`,`T_0`,S>,DelegRecv<`S_0`,T>>`
`DelegRecv<S0,S>`	Accept delegation `S0` then do `S`	`DelegRecv(chan:`p_0`,p)`$^{(2),(3)}$	`Dual<DelegRecv<`S_0`,S>,Deleg<`S_0`,`T_0`,T>>`

*Note: The right half of the table assume that (1) variable `v` has type `Val<V>`, (2) variable `p` has type `Dual<S,T>`, (3) variable p_i has type `Dual<`S_i`,`T_i`>` for $i \in \{L, R, 0\}$.

Figure 3: Session Types and Protocol Combinators

2 Design of `SessionC#`

In this section, we show the design of `SessionC#` which closely follows Honda et al.'s binary session types [8]. § 2.1 introduces protocol combinators, by following Pucella and Tov's approach [25, § 5.2] with a few extensions including recursion and delegation. § 2.2 introduces the improved fluent API, taking inspiration from Scribble [2, 10] and process calculi's literature. § 2.3 discusses an encoding of mutually recursive sessions with less notational overhead.

2.1 Session Types, Protocol Combinators and Duality

Duality is the key to ensure that a pair of session types realise a safe series of interaction. Before introducing protocol combinators, we summarise session types in `SessionC#` in the left half of Figure 3. Type `Send<V,S>` and `Recv<V,S>` are output and input of value of type `V`, respectively, which continues to behave according to the session type `S`. `Select<SL,SR>` means that a process internally *decides* whether to behave according to `SL` or `SR`, by sending either label of `left` or `right`, which is called as *internal choice*. `Offer<SL,SR>` is an *external choice* where a process offers to its counterpart two possible behaviours `SL` and `SR`. `Eps` is the end of a session. `Goto0` specifies transition to the beginning of the session, which makes a limited form of *recursive session*. Later on, we extend this to mutual recursion by having more than one session types in a C# type and accessing them via an index, which is why we annotate `0` as the suffix to `Goto`. `Deleg<S0,T0,S>` is a *delegation* of session `S0` which continues to `S`, where the additional parameter `T0` is the dual of `S0`.

Note that it is possible to implement delegation *without* `Deleg` and `DelegRecv`, but with `Send` and `Recv` instead. The sole purpose of this distinction is the parameter `T0`, which is used by `DelegNew`, which we will develop later, to give the dual session type to the freshly created channel without further protocol annotation. `DelegRecv<S0,S>` is an acceptance of delegation of session `S0` which continues to `S`.

We illustrate our protocol combinators in the right half of Figure 3, making them *prove* duality of two types by restricting the constructors of `Dual<S,T>` to them having `S` and `T` to be dual to each other. In the "Combinator" column of Figure 3, the intuitive meaning of each protocol combinator can be understood

Creating a session

```
var cliCh = prot.ForkThread(srvCh => stmts)
```

Communication

Session type	Method
`Send<V,S>`	`ch.Send(v)`
`Recv<V,S>`	`ch.Receive(out V x)`, `ch.ReceiveAsync(out Task<V> task)`
`Eps`	`ch.Close()`
`Goto0`	`ch.Goto0()`
`Select<SL,SR>`	`ch.SelectLeft()`, `ch.SelectRight()`
`Offer<SL,SR>`	`ch.Offer(left:leftFunc, right:rightFunc)`
`Deleg<S0,T0,S>`	`ch.Deleg(ch2)`, `ch.DelegNew(out Session<T0,T0> ch2)`
`DelegRecv<S0,S>`	`ch.DelegRecv(out Session<S0,S0> ch2)`

Figure 4: The Communication API of `SessionC#`

as the session type in the same row of the left half specifying the *client side*'s behaviour. The "Return type" column establishes duality, by pairing each session type in the first parameter for the client with the reciprocal behaviours in the second one for the server. The type `Val<V>` is the placeholder for payload types of `Send` and `Recv`. For example, `Send(v,p)` with type `Dual<Send<V,S>,Recv<V,T>>` establishes the duality between two session types `Send<V,S>` and `Recv<V,T>` provided that `S` and `T` are dual to each other, which is ensured by the nested protocol object `p`. We defer the actual method signatures of protocol combinators to Appendix § A.

2.2 A Fluent Communication API

In Figure 4, we show the communication API of `SessionC#` which we develop in this subsection. The first column of the figure specifies the session type of the method in the second column. The fluent interface contributes to reducing the risk of linearity violation, by returning the channel with a continuation session type which increase the opportunity to chain the method call. An exception is `Offer` which takes two functions *`leftFunc`* and *`rightFunc`* taking a channel with different continuation session type for selection labels *`left`* and *`right`*, respectively.

2.2.1 Channels and Threads Maintaining Duality

The *channel type* `Session<S,E>` plays the key role in maintaining a session's evolution in the recursive type structure, where the type parameter `S` is the session type assigned to the channel, while `E` is the *session environment* of a channel which serves as a table for recursive calls (`Goto`) to look up the *next* behaviour. In other words, the `S`-part progresses when the interaction occurs on that channel, while the `E`-part *persists* (i.e., remains unchanged) during a session, maintaining the global view of a session. Thus, for example, in a method call `ch.Send(v)`, the channel `ch` must have type `Session<Send<V,S>,E>`, which returns `Session<S,E>`. We explain how the recursive structure is maintained later in § 2.3.

Based on the duality established by the protocol combinators, the `ForkThread` method ensures *safe* communication on `Session<S,E>` channels between the main thread and the forked threads. Concretely, provided `prot` has `Dual<S,T>` saying `S` and `T` are dual to each other, a method call `prot.ForkThread(ch => stmt)`

forks a new server thread, running *stmt* with channel `ch` of type `Session<T,T>`, returning the other end of channel `Session<S,S>`. The `ForkThread` is defined in the following way:

```
class Dual<S,T> {  static Session<S,S> ForkThread(Func<Session<T,T>> fun) { ... }  }
```

Note that the part `<S,S>` (and `<T,T>`) requires the beginning of a session being the same as in the session environment, maintaining the recursive structure by specifying `Goto0` going back to `S` (and `T` resp.).

2.2.2 Protocol Compliance via Extension Methods

The communication API enforces *compliance* to the type parameters in `Session<S,E>`, via *extension methods* of that type which can have additional constraints on type parameters. An extension method is the one which can be added to the existing class without modifying the existing code. For example, the following method declaration adds a method to `List<T>` class in the standard library:

```
static void AddInt(this List<int> intList, int x) { ... }
```

The `this` keyword in the first parameter specifies the method as an extension method, where the possible type of `obj` is restricted to `List<int>`. In this way, we declare the fluent API of output `ch.Send(v)`, for example, as follows:

```
static Session<S,E> Send<V,S,E>(this Session<Send<V,S>,E> ch, V v) { ... }
```

2.2.3 Binders as `out` Parameters, and Async/Await Integration

One of the central ideas of the fluent API in `SessionC#` is to exploit C#'s `out` method parameter to increase chances for method chaining. This is mainly inspired by Scribble [2, 10] implemented in Java, however, thanks to the `out` parameter in C#, there is no need to explicitly passing a *buffer* to receive an input value as in Java, keeping the session-typed program more concise and readable. `Receive` and the acceptance of delegation `DelegRecv` are implemented similarly, in the following way:

```
static Session<S,E> Receive<V,S,E>(this Session<Recv<V,S>,E> ch, out V v) { ... }
static Session<S,E> DelegRecv<S0,S,E>(this Session<DelegRecv<S0,S>,E> ch,
                                      out Session<S0,S0> ch2) { ... }
```

More interestingly, the `out` parameter in the method call `obj.Receive(out var x)` resembles *binders* in process calculi, like an input prefix $a(x).P$ in the π-calculus. By expanding this observation to name restriction $(\nu x)P$ in the π-calculus and other constructs in literature, we crystallise a few useful communication patterns of process calculi in `SessionC#`; namely (1) *bound output* and (2) *delayed input*, where the latter is implemented using async/await.

Bound output is a form of channel-passing where the freshly-created channel is passed immediately through another channel, which is written in the π-calculus as $(\nu x)\overline{a}x.P$, and $\overline{a}(x).P$ in short. As it leaves the *other end* of a channel at the sender's side, we need the *dual* of the carried (delegated) session type, which is why we have both carried type `S0` and its dual `T0` in a delegation type `Deleg<S0,T0,S>`. Thus, delegation `Deleg` and its bound-output variant `DelegNew` is defined as follows:

```
static Session<S,E> Deleg<S0,T0,S,E>(this Session<Deleg<S0,T0,S>,E> ch, Session<S0,S0> ch2) { ... }
static Session<S,E> DelegNew<S0,T0,S,E>(this Session<Deleg<S0,T0,S>,E> ch,
                                        out Session<T0,T0> ch2) { ... }
```

See that `DelegNew` declares the `out` parameter in the second one, where it binds the dual type `T0` of the delegated type `S0`.

```
1 var cliCh2 = cliCh.Send((16, 3, 2))
2                   .DelegNew(out var cancelCh);
3 Task.Delay(10000).ContinueWith(_ => {
4   cancelCh.Send().Close(); });
5 cliCh2.Offer(
6   left: cliCh3 => {
7     cliCh3.Receive(out var ans).Close();
8     Console.WriteLine("Tak(16,3,2) = " + ans);
9 }, right: cliCh3 => {
10    cliCh3.Close();
11    Console.WriteLine("Cancelled");
12 });
```

Figure 5: A `tak` Client with a Timeout

The `ReceiveAsync` is a form of delayed input in the π-calculus literature [21, § 9.3], also inspired by Scribble's *future* [9, § 13.4]. The delayed input *asynchronously* inputs a value i.e., the execution progresses without waiting for delivery of an input value, which blocks at the place it uses the input variable. This is realised by method call `ch.ReceiveAsync(out Task<V> task)` which binds a fresh *task* to variable `task` which completes when the value is delivered. We illustrate the signature of `ReceiveAsync` in the following[2]:

```
static Session<S,E> ReceiveAsync<V,S,E>(this Session<Recv<V,S>,E> ch, out Task<V> v) { ... }
```

Note that the implementation adheres the communication pattern specified in a session type, as the subsequent communication on the same channel does not take place until the preceding reception occurs.

2.2.4 A `tak` Client Example

Based on the communication API shown in this section, including `Offer` and `DelegNew`, we show an implementation of `tak` client in Figure 5, with a timeout. Line 1 sends the three arguments `(16,3,2)` to the server, and Line 2 freshly creates a channel `cancelCh` and send it to the server using bound output `DelegNew`, for later termination request. Lines 3-4 arranges an output of a termination request in 10 seconds (`10000` milliseconds). `Offer` on Line 5 makes an external choice on a channel. The *`left`* case on Lines 6-8 handles the successful completion of the calculation, where the client receives the result `ans` and print it on the screen. The *`right`* case (Lines 9-12) immediately closes the channel and prints `"Cancelled"` on the console.

As we also noted in Introduction, the cancellation request may be disregarded by the server if she has already finished the calculation. Also note that the delegated channel `cancelCh` must be used according to the linearity constraint (of which dynamic checking in `SessionC#` is yet to be implemented though), even if the client does not wish to cancel the calculation. In that case, the client can send a dummy cancellation request *after* it receives the result.

2.3 Recursive Sessions, Flatly

To handle mutually-recursive structure of a session, we extend the session environment to have more than one session type. We extend the notion of duality to the tuple of session types, and provide the protocol combinator `Arrange(p1,p2,..)`, where `p1`, `p2`, ... refers to them each other via `Goto1`, `Goto2`, ... For example, a protocol specification which alternately sends and receives an integer is written as follows:

```
var prot = Arrange(Send(Val<int>, Goto2), Recv(Val<int>, Goto1));
```

Note that the indices origin from one, to avoid confusion in a session environment with the single-cycled sessions using `Goto0`.

[2] Figure 1 uses the overloaded version where payload type `V` is fixed to `Unit`, having `Task` instead of `Task<V>` in the second argument.

The main difference from the one by Pucella and Tov [25] is that, to avoid notational overhead, we stick on *flat* tuple-based representation `(S0,S1,..)` rather than a nested cons-based list `Cons<S0,Cons<S1,..>>`. This also elides *manual* unfolding of a recursive type from $\mu\alpha.T$ to $T[\mu\alpha.T/\alpha]$ encoded as `enter` in [25], resulting in a less notational overhead in recursive session types than [25]. This ad-hoc encoding comes at a cost; the number of cycles in a recursive session is limited because the size of the tuple is limited, we must overload methods since we do not have a structural way to manipulate tuple types – although the maximum size of tuples of 8-9 seems enough for a tractable communication program. Keeping this in mind, the duality proof, `DualEnv<SS,TT>`, which states the duality between the tuple of session types `SS` and `TT`, as well as `ForkThread` and `Goto` methods for mutually recursive sessions are implemented as follows:

```
static DualEnv<(S1,S2),(T1,T2)> Arrange<S1,S2,T1,T2>(Dual<S1,T1> p1, Dual<S2,T2> p2) { ... } ...
static Session<S1,(S1,S2)> ForkThread<S1,S2,T1,T2>(this DualEnv<(S1,S2),(T1,T2)> prot,
                                                   Func<Session<T1,(T1,T2)>> fun) { ... }
static Session<S1,(S1,S2)> Goto1<S1,S2>(this Session<Goto1,(S1,S2)> ch) { ... }
static Session<S2,(S1,S2)> Goto2<S1,S2>(this Session<Goto2,(S1,S2)> ch) { ... }
```

The overloaded versions up to 8-ary is defined in similar way.

Notes on Structural Recursion in C#. A reviewer mentioned that there should be an encoding using recursive generic types in C#. For example, it would be possible to declare the following session type in C#, embodying a recursive session where a sequence of integer is received, and then the sum of them is sent back:

```
class SumSrv : Recv<int, Offer<SumSrv, Send<int, End>>> { ... }
```

Although it is possible to declare such *session types* like above, what we need is a *duality witness* (proof) encoded in C#. Consider a duality relation defined as a class, stating that `Recv<V, S>` is a dual of `Send<V, T>` if `S` is a dual of `T`:

```
class DualRecv<V, Cont> : Dual<Recv<V, ...>, Dual<Send<V, ...>>> { ... }
```

We must refer to the two components of `Cont` in the two ellipsis parts `...`, which would look like the following pseudo-code:

```
// pseudo-C# code!
class DualRecv<V, Cont> : Dual<Recv<V, Cont.S>, Dual<Send<V, Cont.T>>> { ... }
```

which is not possible in C# for now. One might recall *traits* or *type members* in C++ and Scala, and *associated types* in Haskell [4]. There exists an encoding from C#'s F-bounded polymorphism to *family polymorphism* [26], at the cost of much boilerplate code. That said, the use of recursive generic types seems promising, and we are currently seeking a better design for recursive protocol combinators.

3 Application

As a more interesting application of `SessionC#`, we show a *Bitcoin miner*, where a collection of threads *iteratively* try to find a *nonce* of the specified *block*. The protocol for the Bitcoin miner is the following:

```
var prot = Select(left: Send(Val<Block>, Deleg(chan:Recv(Unit,End),
                           Offer(left:Recv(Val<uint>,Goto0), right:Goto0))),
                 right: End);
```

The endpoint implementation is in Figure 6. The `Parallel` method runs multiple threads in parallel, by passing the anonymous function a pair of the server channel `srvCh` and an extra argument `id` which is extracted from the array of parameters `ids`. The client asks (`Select`) a server thread to start the calculation

```
1  var cliChs = prot.Parallel(ids, (srvCh, id) => {
2   for (var loop = true; loop;) {
3    srvCh.Offer(left: cont => {
4     var srvCh2 = cont.Receive(out var block)
5                      .DelegRecv(out var stopCh);
6     stopCh.ReceiveAsync(out Task stop).Close();
7     var miner = new Miner(block, id);
8     while (true) {
9      if (miner.TestNextNonce(out var nonce)) {
10       srvCh = srvCh2.SelectLeft()
11                     .Send(nonce).Goto0();
12       break; // back to Offer() again
13      } else if (stop.IsCompleted) {
14       srvCh = srvCh2.SelectRight().Goto0();
15       break; // back to Offer() again
16      } else { continue; }}},
17    right: end =>
18     { end.Close(); loop = false; });}});
```

Figure 6: A Bitcoin Miner Server

by selecting `left` label, and then it sends a bitcoin `Block` and a cancellation channel in a row. Dually, after the server enters the main loop in Line 2, it offers a binary choice in Line 3, receives the block and a channel in Line 4-5, and then schedules asynchronous reception of cancellation in Line 6. After that, the server starts the calculation in Lines 7-9, entering the loop. Meanwhile, the client waits for the server (`Offer`), and if it sees `left` label, then it receives a nonce of an unsigned integer (`uint`). The corresponding behaviour in the server is found in Lines 10-12, where the server goes back to Line 3. In case another thread finds the nonce, the client asynchronously sends cancellation to the server, which is observed by the server in Lines 13-15, notifying the `right` label back to the client. In the both case, the client returns to the beginning (`Goto0`). If nonce is not found and cancellation is not asked, in Line 16, the server tries the next iteration without interacting with the client. By selecting `right` label at the top, the client can ask the server to terminate, where the server closes the session and assigns `false` to `loop` variable in Line 18, exiting the outer loop.

4 Concluding Remarks

We proposed `SessionC#`, a session-typed communication library for C#. The mainstream languages like C# has not been targeted as a platform implementing session-typed library, where one of the reasons is that the type system of the language is not suitable to implement them — they are less capable than other languages like Haskell, Scala, F# and OCaml, in the sense of having richer type inference or type-classes or implicits. Another reason would be that the type system of C# is considered quite similar to Java's one. We proclaim that the *language features* like `out` variables (and closures) also matters for establishing a safe, usable session communication pattern on top of it, including session intervention, as we have shown in the several examples in this paper.

The typestate approach taken by StMungo by Kouzapas et al. [17] equips session types on top of programming front-end Mungo. Gerbo and Padovani [17] also implements session types in typestate-based encoding via code generation using Java's annotation, enabling concurrent type-state programming a concise manner, at the cost that the protocol conformance is checked dynamically. On the other hand, type-states are sometimes *manually* maintained via variable (re)assignment in `SessionC#`, which weakens the static conformance checking. However, we hope that sticking to the library-based implementation with dynamic linearity checking competes to the aforementioned tools by providing the idiomatic usage of fluent interface.

The techniques and patterns incorporated in improved fluent interface in `SessionC#` is orthogonal to tool support, and we see opportunities to build them in combination with other proposals like Scribble, resulting in a concise multiparty session programming environment. Notably, we see that the session

intervention pattern is also effective in multiparty setting. We observe several instances of the fluent interface in Scribble family, albeit without `out` parameters, in Java [2, 10, 11], Scala [27], Go [3], and F# [22], providing *multiparty session types*. Code completion shown in the Introduction is also available in various implementation in Scribble, and most notably, the work by Neykova et al. [22] integrates Scribble with *Type Provider* in F#. SJ by Hu et al. [12] extends Java with session primitives, and also studies the protocol for session delegation in a distributed setting.

The protocol combinators are highly inspired from Pucella and Tov's encoding of duality [25]. To the author's knowledge, the addition of delegation and recursion to [25] is new. We believe the simplification of recursion adds more readability to programs using protocol-combinator based implementations. Scalas et al. [28] and Padovani [24] implements binary session types based on duality encoded in linear i/o types by Dardha et al. [6]. While it does not require any intermediate object like protocol combinators, we see the encoded session types sometimes have type readability issue, as it makes a nested, flipping sequence inside i/o type constructors, as mentioned in [13, § 6.2]. Imai et al. [13] solved this readability issues via *polarised session types*, at the cost of having polarity in types. Albeit the lack of session type inference in `SessionC#`, we also see the *explicit* approach taken by protocol combinators is not a big obstacle, as it is also the case in C# to declare method signatures explicitly.

Acknowledgements We thank the reviewers for thorough review and helpful comments for improving this paper. This work is partially supported by KAKENHI 17K12662 from JSPS, Japan, and by Grants-in-aid for Promotion of Regional Industry-University-Government Collaboration from Cabinet Office, Japan.

References

[1] *C# documentation*. `https://docs.microsoft.com/dotnet/csharp/`.

[2] *Scribble*. `http://www.scribble.org/`.

[3] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): *Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures*. *Proc. ACM Program. Lang.* 3(POPL), doi:10.1145/3290342.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones & Simon Marlow (2005): *Associated types with class*. In Jens Palsberg & Martín Abadi, editors: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, ACM, pp. 1–13, doi:10.1145/1040305.1040306.

[5] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2006): *Links: Web Programming Without Tiers*. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, pp. 266–296, doi:10.1007/978-3-540-74792-5_12.

[6] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

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

[8] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–138, doi:10.1007/BFb0053567.

[9] Raymond Hu (2017): *Distributed Programming Using Java APIs Generated from Session Types*. In [29, Chapter 13].

[10] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification through Endpoint API Generation*. In: *19th International Conference on Fundamental Approaches to Software Engineering*, *LNCS* 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.

[11] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*. In Marieke Huisman & Julia Rubin, editors: *Fundamental Approaches to Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 116–133, doi:10.1007/978-3-662-54494-5_7.

[12] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pp. 516–541, doi:10.1007/978-3-540-70592-5_22.

[13] Keigo Imai, Nobuko Yoshida & Shoji Yuen (2018): *Session-ocaml: a Session-based Library with Polarities and Lenses*. *Sci. Comput. Program.* 172, pp. 135–159, doi:10.1016/j.scico.2018.08.005.

[14] Keigo Imai, Shoji Yuen & Kiyoshi Agusa (2010): *Session Type Inference in Haskell*. In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.*, pp. 74–91, doi:10.4204/EPTCS.69.6.

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

[16] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou & Alceste Scalas, editors: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, *EPTCS* 304, pp. 48–60, doi:10.4204/EPTCS.304.4.

[17] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking protocols with Mungo and StMungo*. In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pp. 146–159, doi:10.1145/2967973.2968595.

[18] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[19] Sam Lindley & J. Garrett Morris (2016): *Embedding Session Types in Haskell*. In: *Haskell 2016: Proceedings of the 9th International Symposium on Haskell*, ACM, pp. 133–145, doi:10.1145/2976002.2976018.

[20] J. McCarthy (1979): *An Interesting LISP Function*. *Lisp Bull.* (3), pp. 6–8, doi:10.1145/1411829.1411833.

[21] Massimo Merro & Davide Sangiorgi (2004): *On asynchrony in name-passing calculi*. *Mathematical Structures in Computer Science* 14(5), p. 715âĂŞ767, doi:10.1017/S0960129504004323.

[22] Rumyana Neykova, Raymond Hu, Nobuko Yoshida & Fahd Abdeljallal (2018): *A session type provider: compile-time API generation of distributed protocols with refinements in F#*. In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, ACM, pp. 128–138, doi:10.1145/3178372.3179495.

[23] Dominic Orchard & Nobuko Yoshida (2016): *Effects as sessions, sessions as effects*. In: *POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 568–581, doi:10.1145/2837614.2837634.

[24] Luca Padovani (2016): *A Simple Library Implementation of Binary Sessions*. *Journal of Functional Programming* 27, p. e4, doi:10.1017/S0956796816000289.

[25] Riccardo Pucella & Jesse A. Tov (2008): *Haskell Session Types with (Almost) No Class*. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, ACM, New York, NY, USA, pp. 25–36, doi:10.1145/1411286.1411290.

[26] Chieri Saito & Atsushi Igarashi (2008): *The Essence of Lightweight Family Polymorphism*. Journal of Object Technology 7(5), pp. 67–99, doi:10.5381/jot.2008.7.5.a3.

[27] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In: *ECOOP*, doi:10.4230/LIPIcs.ECOOP.2017.24.

[28] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In: *ECOOP 2016: 30th European Conference on Object-Oriented Programming*, *LIPIcs* 56, Dagstuhl, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[29] António Ravara Simon Gay, editor (2017): *Behavioural Types: from Theory to Tools*. River Publisher, doi:10.13052/rp-9788793519817.

[30] Philip Wadler (2014): *Propositions as sessions*. *J. Funct. Program.* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

A Protocol Combinators

The protocol combinators are implemented as the following C# static methods and fields. The ellipsis parts are obvious return statements like "`return new Dual<Send<V,S>,Recv<V,T>>()`":

```
class ProtocolCombinators {
  static Val<V> Val<V>() { return new Val<V>(); }
  static Dual<Eps,Eps> End = new Dual<Eps,Eps>;    static Dual<Goto0,Goto0> Goto0 = new Dual<Goto0,Goto0>;
  static Dual<Send<V,T>,Recv<V,T>>  Send<V,T> (Func<Val<V>> v, Dual<V,T> cont) {...}
  static Dual<Recv<V,T>,Send<V,T>>  Recv<V,T> (Func<Val<V>> v, Dual<V,T> cont) {...}
  static Dual<Select<SL,SR>,Offer<TL,TR>>  Select<SL,SR,T0,T1> (Dual<SL,TL> contL, Dual<SR,TR> contR) {...}
  static Dual<Offer<SL,SR>,Select<TL,TR>>  Offer<SL,SR,T0,T1> (Dual<SL,TL> contL, Dual<SR,TR> contR) {...}
  static Dual<Deleg<S0,T0,S>,DelegRecv<S0,T>>  Deleg<S0,T0,S,T> (Dual<S0,T0> deleg, Dual<S,T> cont) {...}
  static Dual<DelegRecv<S0,S>,Deleg<S0,T0,T>>  DelegRecv<S0,T0,S,T> (Dual<S0,T0> deleg, Dual<S,T> cont) {...}
}
```

Modifiers such as `public` are omitted. Also, there is a small hack: The use of `Func` in the payload of `Send` and `Recv` enables omitting the parenthesis `()` after `Val<V>` in protocol specifications (as in `prot` in § 1).

B More on Recursion and Delegation

Delegation is also extended to handle with mutual recursive sessions:

```
    static Session<S,E> DelegRecv<S1,S2,S,E>(this Session<DelegRecv<(S1,S2),S>,E> ch,
                                             out Session<S1,(S1,S2)> ch2) { ... }
    static Session<S,E> Deleg<S1,T1,S2,T2,S,E>(this Session<Deleg<(S1,S2),(T1,T2),S>,E> ch,
                                               Session<S1,(S1,S2)> ch2) { ... }
    static Session<S,E> DelegNew<S1,T1,S2,T2,S,E>(this Session<Deleg<(S1,S2),(T1,T2),S>,E> ch,
                                                  out Session<T1,(T1,T2)> ch2) { ... }
```

To cope with the delegation in the *middle* of the session, we further extend the communication API for delegation, as follows:

```
    static Session<S,E> DelegRecv<S0,S1,S,E>(this Session<DelegRecv<(S0,S1),S>,E> ch,
                                             out Session<S0,S1> ch2) { ... }
    static Session<S,E> Deleg<S0,T0,S1,T1,S,E>(this Session<Deleg<(S0,S1),(T0,T1),S>,E> ch,
                                               Session<S0,S1> ch2) { ... }
    static Session<S,E> DelegNew<S0,T0,S1,T1,S,E>(this Session<Deleg<(S0,S1),(T0,T1),S>,E> ch,
                                                  out Session<T0,T1> ch2) { ... }
```

It enables a session delegated in the middle of it by having different session types in a session environment, as in `Session<S0,S1>` above.

```
1  protCA.Listen(IPAddress.Any, 8888, srvCh => {
2   using var c = new SessionCanceller();
3   c.Register(srvCh);
4   for (var loop = true; loop;) {
5    srvCh.Offer(srvQuot =>
6     quote.Receive(out var dest).Send(90.00m)
7          .Offer(srvAcpt => {
8      var cliCh = protAS.Connect("1.1.1.1", 9999);
9      c.Register(cliCh);
10     cliCh.Send(dest)
11          .Receive(out var date).Close();
12     srvAcpt.Send(date).Close();
13     loop = false;
14    }, srvReject => {
15     srvCh = srvReject.Goto();
16    }), srvQuit => {
17     srvQuit.Close();
18     loop = false; });}});
```

Figure 7: A Travel Agency (Agency Part)

```
1  // Client implementation (main thread)
2  foreach (var block in Block.GetSampleBlocks()) {
3    // Send a block to each thread
4    var ch2s = ch1s.Map(ch1 =>
5      ch1.SelectLeft().Send(block));
6    //  external choice
7    var (ch3s, cancelChs) = ch2s.Map(ch2 => {
8      var offer = ch2.DelegRecv(out var cancalCh)
9        .OfferAsync(some => {
10         var _ch3 = some.Receive(out var nonce);
11         return (_ch3.Goto(), nonce);
12       }, none => {
13         var ch3 = none.Goto();
14         return (ch3, default(uint?));
15       });
16     return (offer, cancalCh);
17   }).Unzip();
18   // Wait for any single thread to respond
19   await Task.WhenAny(ch3s);
20   // Send cancellation to each thread
21   cancelChs.ForEach(ch => ch.Send().Close());
22   // Get channels and results from future object
23   var (ch4s, results) =
24     ch3s.Select(c => c.Result).Unzip();
25   // Print results (omitted)
26   // Assign and recurse
27   ch1s = ch4s;
28 }
29 // No blocks to mine, finish channels
30 ch1s.ForEach(ch1 => ch1.SelectRight().Close());
```

Figure 8: A Bitcoin miner client

C More Examples

Figure 7 is an implementation of a Travel Agency from [12], which incorporates two sessions in a distributed setting. The *canceller* in Line 2 declared `using` modifier stops the *registered* sessions in Lines 3 and 9 when scoping out, which enables to propagate connection failure in one of underlying TCP connections to the other.

We leave a few more examples for curious readers. Figure 9 is a *parallel http downloader* from [3], which utilises `Parallel` method defined on the protocol specification object. Figure 8 is a client to Bitcoin miner shown in § 3. Figure 10 is an implementation of *parallel polygon clipping* from [25], where `Pipeline` creates a series of threads connected by two session-typed channels of which session type is described in a protocol specification.

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using Session;
using Session.Threading;
using static ProtocolCombinator;

public class Program {
  public static async Task Main(string[] args) {
    // Protocol specification
    var prot = Select(left: Send(Val<string>,
      Recv(Val<byte[]?>, Goto0)), right: End);

    var n = Environment.ProcessorCount;
    var ch1s = prot.Parallel(n, ch1 => {
      // Init http client
      var http = new HttpClient();

      // Work...
      for (var loop = true; loop;) {
        ch1.Offer(left => {
          var ch2 = left.Receive(out var url);
          var data = Download(url);
          ch1 = ch2.Send(data).Goto();
        }, right => {
          right.Close();
          loop = false;
        });
      }

      // Download function
      byte[]? Download(string url) {
        try {
          return http
            .GetByteArrayAsync(url).Result;
        } catch {
          return null;
        }
      }
    });

    // Pass jobs to each thread
    var (ch2s, ch1s_rest, args_rest) =
      ch1s.ZipWith(args, (ch1, arg) => {
        var ch3 = ch1.SelectLeft().Send(arg)
          .ReceiveAsync(out var data);
        return (ch3.Sync(), data);
      });

    // Close unneeded channels
    ch1s_rest
      .ForEach(c => c.SelectRight().Close());

    var (working, results) = ch2s.Unzip();
    var working_list = working.ToList();
    var result_list = results.ToList();

    // Wait for a single worker finish
    // and pass a new job
    foreach (var url in args_rest) {
      var finished =
        await Task.WhenAny(working_list);
      working_list.Remove(finished);
      var ch3 = (await finished).Goto()
        .SelectLeft().Send(url)
        .ReceiveAsync(out var data);
      working_list.Add(ch3.Sync());
      result_list.Add(data);
    }

    // Wait for still working threads
    while(working_list.Any())
    {
      var finished =
        await Task.WhenAny(working_list);
      working_list.Remove(finished);
      (await finished).Goto()
                      .SelectRight().Close();
    }

    // Save to files or something...
  }
}
```

Figure 9: Parallel HTTP Downloader [3]

```
using System;
using System.Collections.Generic;
using Session;
using Session.Threading;
using static ProtocolCombinator;

public class Program {
  public static void Main(string[] args) {
    // Input: clippee
    var vertices = new Vector[] {
      new Vector(2.0, 2.0),
      new Vector(2.0, 6.0),
      // ... and more points
    };

    // Input: clipper
    var clipper = new Vector[]
    {
      new Vector(1.0, 3.0),
      new Vector(3.0, 6.0),
      // ... and more points
    };

    // Split clipper each edges
    var edges =
      new (Vector, Vector)[clipper.Length];
    for (int i = 0; i < edges.Length; i++) {
      edges[i] = (clipper[i],
        clipper[(i + 1) % clipper.Length]);
    }

    // Protocol specification
    var prot = Select(left: Send(Val<Vector>,
      Goto0), right: End);

    var (in_ch, out_ch) = prot.Pipeline(edges,
      // Each thread
      (prev1, next1, edge) => {
        Vector? first = null;
        Vector from = default;
        Vector to = default;
        for (var loop = true; loop;) {
          prev1.Offer(left => {
            var prev2 = left
              .Receive(out var vertex);
            from = to;
            to = vertex;
            if (first == null) {
              first = to;
            } else {
              var clipped =
                Clip((from, to), edge);
              foreach (var v in clipped) {
                next1 = next1
                  .SelectLeft().Send(v).Goto();
              }
            }
            prev1 = prev2.Goto();
          }, right => {
            var clipped =
              Clip((to, first.Value), edge);
            foreach (var v in clipped) {
              next1 = next1.SelectLeft()
                           .Send(v).Goto();
            }
            next1.SelectRight();
            loop = false;
          });
        }
      }
    );

    // Main thread
    // Send vertices to pipeline
    foreach (var v in vertices) {
      in_ch = in_ch.SelectLeft().Send(v).Goto();
    }
    in_ch.SelectRight().Close();

    // Collect result from pipeline
    var result = new List<Vector>();
    for (var loop = true; loop;) {
      out_ch.Offer(left => {
        out_ch = left.Receive(out var vertex)
                     .Goto();
        result.Add(vertex);
      }, right => {
        right.Close();
        loop = false;
      });
    }

    // Print result
    for (int i = 0; i < result.Count; i++) {
      Console.WriteLine(result[i]);
    }
  }
}
```

Figure 10: Polygon Clipping Pipeline [25]