AXUM

LANGUAGE OVERVIEW

Niklas Gustafsson
v0.8

Microsoft

Orchestrating Agents

This document covers a proposed agent-oriented special-purpose language referred to as codename "Axum." The primary use for Axum is to define concurrent application logic of relatively coarse granularity. Axum borrows its syntactic style from the C family, most closely C#, with which it shares most of its expression and statement language elements.

The message-passing concepts of Axum borrow heavily from prior prototypes and product efforts within and outside Microsoft. The programming languages Ada, and SR, were also inspirations.

Overview

C# and Axum have the following language elements in common:

- All C# 3.0 expressions, including lambdas and LINQ[1] queries
- All C# 3.0 statements, including yield return and yield break
- Method and field declarations
- Delegates and enumeration types

The following C# language elements are not present in Axum:

- Classes, interfaces and structs
- Operator declarations
- Properties
- Const fields and const locals
- Static fields and methods

The language features unique to Axum can be grouped into the following high-level categories:

- Agents and domains
- Channels
- Schema
- Networks
- Interleaved control flow

Of these, the first three are primarily concerned with state isolation and information exchange between isolated regions of an application. The last two concern the message orchestration via data and control flow, respectively.

An *agent* is the most basic isolation concept in Axum: each agent is similar to a class, but a reference to the agent instance cannot be held anywhere. Instead, agents interact with each other via message-passing over

[1] Specified, but not implemented in the Dev Labs version of Axum.

separately defined channels. These channels define discrete *ports* through which data passes. Channels may also define formal protocols for the legal exchanges of data between communicating parties.

A *domain* is the next larger isolation concept in Axum: it may be viewed as a class with only private fields and methods, isolated from all other domains: only constructors are accessible outside the domain definition. Agents can be nested within a domain, in which case the agents have access to the domain state but their accesses are automatically orchestrated to prevent data races.

To define the data that passes between domains, agents and applications, Axum allows a special "data transfer" type definition, called a *schema*. Similar to the schema used for an XML document passed between two web services, a schema defines the structure and rules for data passed between isolated components, i.e. agents.

Central to the interaction of agents is the concept of messaging, i.e. the passing of values between components asynchronously or synchronously. Through receive statements and interleaved control flows, handling messages in a control-flow-like structure is straightforward.

Axum defines the abstract concept of and interaction point and an approach to build *networks* from such interaction points, allowing the definition of a medium-grained[2] dataflow model for message processing within an agent or domain. Channels participate in this: channel ports are interaction points.

The following sections cover the additional language elements one by one.

[2] Meant to indicate something in between the granularity you would use calling methods on objects (fine-grained) and remote procedure calls (coarse-grained).

Channels

Before we can discuss agents, we need to cover the semantics of channels in some detail. In a later section, channels are specified more thoroughly.

Channels are types presenting a collection of ports: each port supports asynchronous unbounded buffering of data with strict in-order semantics across all ports of a channel. They may be used for connections across logical or physical isolation boundaries: domain, process, or machine boundaries; Axum channels are designed to be compatible with WCF channels, enabling distributed communication using the same agent-based programming model.

Ports that allow data to be sent to them are called *input* ports. Ports from which data can be received are called *output* ports. When two parties are communicating on a channel, the data is sent to the input port of a channel, and received from the corresponding output port on the other side of the channel. That is, every port acts as input or output port depending on the point of view of the party communicating on the channel.

Each channel instance is thus represented by two channel endpoints, one on each side of the isolation boundary that the channel is intended to cross. As a convention, we say that one endpoint has a 'using' perspective on the channel, while the other has an 'implementing' perspective; this convention is necessary in order to allow us to reason about the asymmetric aspects of channels.

The using and implementing perspectives are significant enough that declarations for each side are of different types – given a channel A, the using-side type is 'A' and the implementing side 'A.implements.' There is, in other words, a strong type concept in place for channel endpoints which prevents mismatched perspectives

Channel endpoints are created and used as proxies, i.e. the implementation details of the link between its two endpoints is not known.

Normally, a connection is established from the "using" side, by a channel constructor, supplying an address of the implementing side ('Channel_A' is here assumed to be some channel we have declared elsewhere):

```
var chan = new Channel_A("TestService");
```

The type or format (contents) of such addresses and what it is that they represent is entirely outside the language definition: Axum anticipates the concept and applies it to communication with agents, but addresses are otherwise entirely abstract within the language. The only constraint placed on addresses is that they must have an identity that is separate from their object identity, i.e. value-equality rather than reference-equality. That said, strings are likely to be the most commonly used address form, given how convenient they are when manipulated by humans.

Ports

Ports are the most fundamental elements of a channel definition. Each port represents a logical buffer, inbound or outbound, with some program-specific semantics attached to it.

A channel port definition consists of an orientation, a name for the port, and a type. A port may also be declared to have two types, effectively declaring a two-way port to support the very common request / reply message exchange pattern.

Channels can use a constrained set of the general .NET type system: only immutable or fully serializable types can be used in a port. Examples of such types are int and string, as used in the example below. Later, we will discuss the full set of types usable in channel communication, under the coverage of 'Schema' and 'Channels Revisited.'

Example:

```
channel Channel_A
{
  input  Pair<int,int> Add;
  output int Sum;
}
```

This sample defines a channel with an inbound port named 'Add' and an outbound port named 'Sum.' The keywords 'input' and 'output' indicate port orientation: the direction in which values are transferred via a port: 'input' means that values are transferred from the using side to the implementing side of the channel, while 'output' means the exact opposite.

All the ports of a channel may be bound together in more or less complex patterns of interactions. Of the possible patterns, request / reply is such an important and common sub-pattern that it is supported explicitly by the port definition syntax. Request / reply is the pattern in use for nearly all synchronous remote protocols used in DCOM, RPC, WCF, etc., where it's most common use is to model a remote procedure call.

Thus, if the channel declaration above was meant to indicate that a couple of integers are to be sent on the Add port, after which the sum is available on the Sum port, then a more tightly coupled definition may be expressed as:

```
channel Channel_B
{
   input Pair<int,int> Add : int;
}
```

This defines a port of type int that is intrinsically tied to the Add port and allows specific reply values to be correlated with specific requests. The reply port has the orientation opposite that of the port it is tied to[3].

[3] See 'Send' for more in-depth coverage of the use of reply ports.

Ports are selected using a specific selection operator '::' instead of '.' This allows us to use collections of channel endpoints on the left of the selection operator and end up with a collection of ports:

```
var chan1 = new Channel_A(...);
var chan2 = new Channel_A(...);

var x = receive(chan1::Sum);
chan2::Sum <-- x;

Channel_A [] arrA = ...;
arrA::Sum >>- chan1::Sum;
```

The operators '<--' and '>>-' used in this example are data-flow network operators that will be discussed in detail later on. For this example, it is only necessary to know that '<--' (send) takes a target interaction point and a single value, while '>>-' (multiplex) takes an enumeration of source points and a single target.

Patterns

A channel that only contains port declarations allows any port to be used at any point in during its lifetime. This means that any port may be the first one to be used and any port can be the last. This freedom is often useful, but does not allow channels to enforce a specific protocol.

If a channel declaration is seen as providing a vocabulary for a conversation between two parties, it may also be necessary to define in what order certain things may be said, as well as defining how the conversation is ended. This defines a grammar to go with the vocabulary, forming a little mini-language for a channel.

Without such a definition, a conversation is ended by convention and the convention may be encoded in conflicting ways on either side of the channel: A thinks that the exchange is done, while B waits for another value.

Channel patterns express a protocol state machine in terms of port usage:

```
channel Channel_A
{
  input  Pair<int,int> Add;
  output int Sum;

  Start: { Add -> S1; }
  S1: { Sum -> End; }
}
```

The pattern consists of a set of states, each of which contains a pattern-match expression yielding a next state.

In the example above, the Start state allows a single transition triggered by the port 'Add' being used, leading to the state named 'S1.' The latter state also allows a single transition, that triggered by the port Sum being used, leading to the End state.

All channel protocols start in the state named ‘Start,’ which is predefined by the Axum language.

The state named ‘End’ is, like ‘Start,’ also predefined. It is the final state of all patterns, in which further use of any port is invalid. Disposing of a channel endpoint in any other state than ‘End’ is a program error.

Any transition that is not specifically allowed in the pattern results in a protocol violation, reported via a ‘ContractViolationException.’ Transitions may also be conditional, taking into account the value of the sent data. For example, we can disallow division-by-zero in the contract below:

```
channel Channel_A
{
  input  Pair<int,int> Add;
  input  Pair<int,int> Divide;
  output int Result;

  Start: { Add -> S1; Divide $ (value != 0) -> S1;}
  S1:    { Result -> End; }
}
```

Channels may define functions to be used in such filter expressions. This is discussed in further detail later on.

Domains

Every Axum program consists of at least one domain, within which the main entry point of an Axum application is declared as an agent implementing the channel type 'Application[4]:'

```
public domain MainDomain
{
  public agent MyMainAgent : channel Application
  {
    public MyMainAgent()
    {
        var cmdArgs = receive(primary::Start);

        Console.WriteLine("Hello World!");

        primary::Done <-- 0;
    }
  }
}
```

When starting such an application, an initial domain and an agent instance are created by the system. The program terminates when a program exit code is sent to the agent's Done port, which is accessible via the agent's primary channel. The primary channel, discussed in further detail under 'Agents,' is always available as the identifier 'primary' within the body of an agent.

The main purpose of domains is to provide explicit isolation of memory between concurrent application components. Any given domain instance will share no variables with any other domain instance created within the Axum model.

The only public members of domains are their constructors, which can only take the same kind of constrained payload types as channel ports. These constraints are discussed below under 'Channels Revisited,' but all value types are included in the set, as is 'String.'

Domains are also created as a side-effect of calling the 'Create()' member of a visible agent type. A new agent instance and a new domain instance are created and connected. The result of creating an agent instance is a using-side endpoint of the channel that the agent implements – the channel is the only way to communicate information from the creating domain to the created domain containing the new agent instance:

```
var main2 = MyMainAgent.Create();
main2::Start <-- new String [] { };
```

Variables, methods and functions declared within a domain may only be referenced (called, in the case of methods and function) from within that domain.

[4] This is actually an example applicable to console applications – depending on a different application model, other channel types may be used to start an application.

While domains can be used to process data in safe isolation from other components of an application, they are rather static. In order to do anything interesting with them, we need to rely on agents, which can be created directly from the agent factory (Agent.Create()) or via hosting by the domain, which is described later.

Agents

Whereas low-level parallelism is often best realized in a shared-memory environment, some applications need parallel structure at a higher level of the architecture, which is what agents provide the Axum programmer.

In Axum, an agent is a contained algorithm which cooperates with other agents using message passing within or between domains. It is logically similar to a thread within an operating system process (domain), but is much more light-weight than a thread. There can be hundreds of thousands or even millions of simultaneous agents in a program.

Each agent must implement a single channel type (discussed later), which is accessed via the identifier specified in a 'channel' clause:

```
public agent Main : channel Application
{
}
```

Agents should define either no constructor, or a simple parameter-less constructor, which is where control starts after the agent is associated with its containing domain. The constructor may contain logic spanning the entire lifetime of the agent instance, of just code to set up data-flow networks:

```
public writer agent Main : channel Channel_C
{
  public Main()
  {
      bool done = false;

      while (!done)
      {
        var cmd = receive(primary::Command);

        if (cmd == null)
            done = true;
        else
            Console.WriteLine(cmd.Name);
      }
  }
}
```

Simple network construction often requires no explicit constructor:

```
public writer agent Main : channel Channel_C
{
  private IDisposable nwk =
    primary::Command ==> (cmd =>
    {
      if (cmd == null)
        nwk.Dispose();
      else
```

```
            Console. WriteLine(cmd.Name);
        });
    }
```

Hosting

Each agent instance is associated with a specific single domain instance – we've already seen how the initial agent instance of an application is created by the runtime and how new agent instances can be created with a new domain instance by using the 'Create' method defined by all agents.

While this method of creating agents is very convenient and efficient, it is also very limited. There are primarily two things it doesn't let you do:

1. Create a new agent instance within an existing domain instance.
2. Create an agent instance in another process.

The first presents an issue when multiple "clients" of the domain all want to access the domain state through an agent – since each client will be creating a new domain instance, there's no actual sharing of state. The second is an issue when we are trying to scale the Axum programming to distributed uses, something its isolation model is ideal for.

A more general method for creating agents is available through the 'hosting' API, and it requires a little more code than the more limited version. Generally speaking, an Axum host is a component that has enough runtime access and knowledge to create agent instances on behalf of clients. The host must be implemented outside the program in a language such as C# or VB.

The term 'hosting an agent,' commonly used in documentation, refers to the act of asking a host implementation to create a factory for agent instances given an agent type and an address. Typically, addresses are URI-formed strings, but there is no requirement that this be the only format. 'Evicting' an agent refers to the act of asking the host to remove the factory associated with an address.

All hosts implement IHost, which defined in the Axum runtime; there are two built-in hosts: InprocHost and WcfServiceHost.

Example hosting an agent at a given string address for two hours:

```
public domain X
{
  public X(string address)
  {
    IHost hst = new InprocHost();
    hst.Host<Proc>("a");

    Timer(TimeSpan.FromHours(2)) --> (x => { hst.Evict("a");} );
  }

  private reader agent Proc : channel CtLessSimple
  {
```

```
    public Proc()
    {
      // Do something meaningful.
    }
  }
}
```

Hosts allow programmers to call the IHost members directly, but due to the fact that the code cannot access the domain reference, there is no way to associate an agent factory created that way with a domain instance. Thus, all agents hosted by direct calls to the IHost members will result in a new domain instance created for each new agent instance, just like when using the 'Create' method.

To help with this, all domains support an API 'Host<T>' to which is passed an IHost reference, an address, and a creation mode. It is recommended that this be used instead of calling the IHost members directly:

```
public domain X
{
  public X(string address)
  {
    IHost hst = new InprocHost();
    Host<Proc>("a", AgentCreationMode.UseExisting, hst);

    Timer(TimeSpan.FromHours(2)) --> (x => { hst.Evict("a");} );
  }

  private reader agent Proc : channel CtLessSimple
  {
    public Proc()
    {
      // Do something meaningful.
    }
  }
}
```

Agent instance Creation

Except for domain-level instances, where the agent type's Create() method is used, agent instances are created by a client[5] creating a channel from its address and sending it a value via one of its inbound ports. This model for instance creation means that the client will always send the first message, so the channel type in use must allow this in its protocol.

If the agent is hosted at the address when the message arrives, an instance of the agent is initialized and its parameter-less constructor is invoked. Should the agent not be hosted at the address, or the construction code thrown an exception, the channel is faulted and its ports cannot be used further.

[5] The term *client* is here used loosely to denote any component that is initiating contact with another; the latter is herein referred to as the *service*.

Channel creation is done by using an instance of ICommunicationProvider corresponding to the transport used to access a hosted agent. For example, if the agent is to be accessed via WCF, the WcfCommunicationProvider should be used. In the following example, we're using an InprocCommunicationProvider to connect to the 'Proc' agent in the previous section:

```
var cmp = new InprocCommunicationProvider();
var chan = cmp.Connect<CtLessSimple>("a");
```

Some transports may require that a message be sent from the using side of the channel before an agent instance is actually created. For example, using the WcfCommunicationProvider to connect will defer agent creation until the client sends a message to the server. Channels where the first message is coming from the implementing side (i.e. using an output port) will not work correctly with such providers.

Inside the Agent Body

Within the agent instance, the identifier 'primary' refers to the implementing endpoint of the channel of the agent. In the declaration

```
reader agent Helper : channel CtSimple { ... }
```

The type of 'primary,' available throughout the body of the agent, is actually 'CtSimple.implements,' a type representing the implementing side of the channel. Only rarely does a programmer have to be aware of this irregularity: one scenario may be when storing implementing-side endpoints in domain variables, which may need explicit type declarations. In the following example, an agent B is handling all interactions with clients one by one, while agent A serves only accept incoming connections and forward them to B:

```
public domain X
{
  public X() { B.Create(); }

  private OrderedInteractionPoint connections =
      new OrderedInteractionPoint<CtA.implements>();

  private agent A : channel CtA
  {
    public A() { connections <-- primary; }
  }

  private agent B : channel Empty
  {
    private Process(CtA.implements p)
    {
        // Process client requests
    }

    public B()
    {
        host A "internal:X.A";
```

```
            while (true)
              Process(receive(connections));
        }
      }
    }
```

Agents receive inbound messages from the primary channel and send outbound. In addition to the primary channel, an agent may create channels connecting to other agents and thus act as clients to them.

Agents may declare fields, methods, and functions, which are all instance members of the agent (there are no static members). They are 'protected' unless explicitly marked as 'private.' An agent may also be sealed, in which case other agents cannot be derived from it.

Members of agent types do not specify an identifier to refer to the agent instance – all member references are via implicit reference to the agent instance.

Agent Inheritance

An agent may inherit from another agent: the protected methods of the base agent are virtual and may be called from within the derived agent.

At most one base agent may be used, but a new channel type can also be defined on the derived agent. The channel must inherit from the base agent's channel type in a conforming way[6]:

```
public agent A : channel CtA
{
    public A() { Run_1(); }

    Run_1() { ... }
}

public agent B : A, channel CtB
{
    override Run_1() { ... }
}
```

Sharing Domain Members

By virtue of having access to variables declared within domains that contain them, agents can share state and interact outside the synchronization model that channels provide.

In order to access mutable domain variables, agents must declare themselves readers or writers: 'reader' and 'writer' specifiers are added before the 'with' clause:

```
public domain X
{
```

[6]Conformance is discussed in detail further down under 'Channels Revisited'

```
  private int i = 10;

  public writer agent A : channel CtA
  {
    int a = receive(primary::MsgA);
    i += a;
  }

}
```

- Agents that have neither 'reader' nor 'writer' specifiers can only access the domain's immutable variables.
- Any agent that declares itself to be a 'reader' has access to the domain's immutable variables, functions and has read access to any mutable fields and properties. It can also send and receive messages to / from any source or target reference (see the discussion of networks later on) declared in the domain.
- Any agent that declares itself to be a 'writer' has full access to the domain's members.
- Accessing object types that are not immutable (such as most .NET framework types) requires writer access if it is possible that the reference came from a domain variable[7].

Parent Reference

Agents typically access the members of a containing domain directly, without any qualifying reference. However, when names collide, or when it is useful to be explicit about the domain reference for code readability, domain members can also be accessed via the 'parent' reference, a predefined identifier within agent bodies.

[7] This is a very unfortunate consequence of using mutable types in Axum.

Networks

When processing data passed asynchronously, it is usually necessary to buffer the data so that a recipient can take its time to prepare for the arrival of the data, or moderate the rate at which it is accepted. Buffering can take many forms, from an unbounded queue to a mutable variable that holds no more than one value to a write-once buffer that can only be filled with data once.

Axum does not try to anticipate all the possible ways that data may need to be buffered, nor does it lock down the semantics of routing data in and out of such buffers. Instead, it relies on the notion of abstract *interaction points* that may be composed into data flow networks or used directly in asynchronous operations.

Interaction Points

Interaction points form the basis of all asynchronous operations in Axum. The concept presents the programmer with a compositional model for routing asynchronously communicated data, providing the means to safely extend the set of available messaging primitives.

This document shall not go into detail on the exact nature of the interaction points APIs involved. We should only point out to that interaction points implement at least one of ITarget<a> and ISource<a>, with most interaction points implementing both.

We also define some interaction points that implement ITarget<a0> and ISource<a1> where a0 and a1 may be different. For example, the Transform interaction point does this. As we saw in the section on 'network expressions' above, interaction points are often constructed by expressions directly in the code – the concrete type is then defined by the runtime.

Interaction points are integrated into Axum due to the expressive power they offer the language: making sure that the infrastructure routes the right values to the right places is important if we are to build a simple, flexible and robust asynchronous program. It allows data processing to be defined in terms of asynchronous workflow (receive) as well as data flow (connecting interaction points together with sinks and transforms).

The simplicity comes from the lack of complexity of the primitives; the flexibility from the general nature of the pattern; the robustness from the well-known semantics of each building block. This approach to composition has worked well in UNIX shell programming for decades, but software networks have also been used to program IP routers and in embedded systems.

Unlike, for example, Kahn process networks, Axum's concept of a network makes no formal definition of how processing nodes behave – extensibility of the set of available buffering elements (i.e. network routing elements) is a specific objective of the design and prevents formal definition.

Networks are based on a shared memory model, and are thus only useable within a single domain, i.e. to route data between agents within a single domain instance.

Networks

Networks are combinations of interaction points into more complex message routes, abstracting or hiding the constituent components. They are formed using seven binary operators and one unary operator:

- **Forward** (==>), which will link its left operand to its right operand so that the left operand will attempt to propagate data sent to it to the right operand. The link exists until the expression is disposed or explicitly disconnected.
- **Forward once** (-->), which links its operands in the same manner and is similarly disposed, but which automatically disconnects as soon as one message is successfully propagated from the left to the right operand. Note the similarity to the operator '<--', which represents an originating send operation.
- **Multiplex** (>>-), which links a collection of left-hand operands to a single right-hand operand in the same manner as 'forward.' All connections are established at once and disconnected together.
- **Combine** (&>-) , a.k.a. 'Join', which accepts a collection of interaction points as its left-hand operand and produces an array of data, one value from each interaction points per message. Also, for a left-hand operand that is a tuple of interaction points, the expression will produce a tuple of values.
- **Broadcast** (-<<), which accepts a single interaction point on the left and a collection of interaction points on the right; it propagates all data from the left operand to all the interaction points on the right.
- **Alternate** (-<:), which propagates data from the single left operand to the interaction points in the right-hand collection in round-robin order.
- **Tuple** [8](,,,), which forms a network with the left-hand operand as the visible target and the right-hand operand as the visible source. The two are not linked together: the left operand does not have to also be a source, and the right operand does not have to be a target.
- **Filter**[9] (???), a unary operator which will create a buffer propagating data that passes the 'a -> bool' filter function that is the operand of the filter.

Functions interact with network expression by acting as interaction points: a function of type 'a -> b' may be used as a buffer between a source producing values of type a and a target accepting values of type b:

```
var xa = new OrderedInteractionPoint<int>();
var xb = new OrderedInteractionPoint<string>();

xa ==> (x => x.ToString()) ==> xb;
```

Each interaction point may introduce parallelism as it sees fit, but any interaction points that execute user code on a separate thread must participate in the reader / writer regimen vis-à-vis domain and agent state.

[8] The tuple operator is currently not implemented in the compiler.
[9] The filter operator is currently not implemented in the compiler.

A simple pipeline can thus be constructed:

```
var xc = new OrderedInteractionPoint<int[]>();
xc ==> (arr => arr.Sum()) ==> (i => i.ToString()) ==>
  ??? (str => String.IsNotNullOrEmpty(str)) ==> xb
```

If we now send an array of 1 through 4 to xc:

```
xc <-- { 1; 2; 3; 4 }
```
[10]

we should see the string "10" end up in the target buffer. Note that if we keep sending data through this pipeline, the stages can start processing data fully in parallel since none of the transforms change any state.

Network Declarations

While the expressions above are sufficient to build complex and powerful data-flow networks, they don't hide any of the details and they are not re-usable: each construct defines a single instance. To support hiding the internal routing details of a given network as well as allowing such networks to be re-used in many places, Axum supports network declarations.

Such networks are very similar to classes, except that the only public members are the constructors and the ports of the network. Everything else, i.e. fields, methods and functions, are either private or protected members.

Like channels, networks have ports and may define patterns of how the ports must be used together. Ports are directional, with input ports used to send information into the network and output ports to get it out. Request / reply ports are used exactly as with channels. Unlike channels, there are not two usage perspectives on networks – they are single-instance. Using-side channel endpoints can be thought of as networks where all the internal details are compiler generated (that is, in fact, what they are).

While offering no isolation mechanisms, networks are very core to the Axum language. Most asynchronous libraries are[11] modeled as networks rather than C# classes. In many ways, networks are asynchronous classes: they can hold state and provide operations via port interactions.

Our first sample network is functional, transforming a string using a function passed in to the constructor:

```
public network TransformString
{
  input string in = func;
  output string out = a;

  public TransformString(Func<string,string> func) { expr = func; }
```

[10] This array-construction syntax is not currently supported by the compiler. Use C#-style array creation instead.
[11] This is the aspiration. There are no libraries built for Axum at the moment. ☺

```
  private Func<string,string> expr;

  private InteractionPoint<string> a = new OrderedInteractionPoint<string>();

  layout { expr ==> a; }

}
```

This network has two ports, 'in' and 'out', which are bound to internal constructs: the input is bound to a transform function, while the output is bound to a buffer (a common pattern). The constructor accepts the transforming function as an argument and stores it in 'expr.' Last in the network, there's a layout block where all the internal routing of the network is done using the same network operators that were introduced previously.

The network may be used in another expression:

```
var trsfm = new TransformString(x => x.ToLower());

xString ==> (trsfrm::in ,,, trfrm::out) ==> yString;
```

Since there is exactly one input port and exactly one output port, the ports can be defined as default ports, which influences the type and use of the network. Default ports are not named and their type is inferred from the expression they are bound to in the port declaration. There can be at most one default input port and one default output port per network declaration.

Using default ports, our network would look like:

```
public network TransformString
{
  input = func;
  output = a;

  ...
}
```

In this case, the type 'TransformString' implements IInteractionSource<string> (by having a default string output port) and IInteractionTarget<string> (by having a default string input port) and can be used directly in a network expression:

```
var trsfm = new TransformString(x => x.ToLower());

xString ==> trsfrm ==> yString;
```

Networks may be generic.

Faults

Since Axum is a language meant to interact with other languages it needs to provide bridges for concepts that are inconsistent with the model itself. One such need is presented by exceptions, i.e. auto-propagated error

descriptors objects following a stack-based discipline for associating error handlers with specific categories of errors.

This stack-based discipline transfers poorly to the asynchronous world, where program stacks are not necessarily able to provide a good context for error handling. This is particularly true for propagation of messages within a network.

Networks communicate any exception that escape from transforms and sinks or are otherwise raised during message propagation via a port 'Faults,' which is present on all networks, whether declared or simply formed by network operators. This is done by turning an exception into a message and making it available on a fault port that all networks have.

The built-in network port is called 'Faults' and is of type 'Microsoft.Axum.Fault.' The payload type is a wrapper of exception objects that merely adds some information about which source and target were involved with the exception and changes the conceptual name to avoid the synchronous connotations that 'Exception' has in .NET.

Exceptions raised within an agent are likewise reported as a Fault instance to the client of the agent. The channel is immediately moved to the 'End' state and no longer usable for communication.

Empty / Full Storage

Axum defines three forms of 'asynchronous' storage, are modifiers for local variables, formal arguments and fields of domains, agents, and networks.

The three forms, variants of the classic empty/full pattern, are:

- 'const' storage, which accepts a value once and does not let it be overwritten. Initially empty, it is forever full once written to.
- 'async' storage, which accepts a value and lets it be overwritten any number of times. Initially empty, it is forever full once written to.
- 'sync' storage, which accepts a value, blocks writers if full and empties when read.

Storage is specified as part of the declaration:

```
const int x;
async int x;
sync  int x;
```

Note the syntactic peculiarity of the 'const' version – an uninitialized const! While const-storage variables behave a lot like C# consts, there is no need to initialize it at the point of its declaration: anyone attempting to read it will block until it is defined.

All variables, fields, and formals with asynchronous storage are treated as sources and targets. This means that they can be used in network expressions.

Asynchronous storage is very important for integrating the Axum data-flow constructs with control-flow statements and expressions. It is sometimes desirable to "form the shape" of a computation before having all the inputs available, especially when the usual data-flow blocks are insufficient. For example, reading two input ports with price information, finding the lowest value, and sending the output to a port could be done thus:

```
private IInteractionSource<decimal> min(async decimal left, async decimal right)
{
    async decimal x;
    var min_func = (i => Math.Min(i[0], i[1]));
    {left,right} &>- min_func --> x;
    return x;
}

public Agent_1()
{
    min(chan_1::Price, chan_2::Price) --> primary::Result;
}
```

The method 'min' builds a network to produce the value into asynchronous storage, which is returned as a source. The actual computation is delayed until the input values are available.

An asynchronous variable, field, or argument is automatically waited for when used in a synchronous context, such as when storing it to a regular variable, or passing it to a function that does not use asynch storage. Likewise, a regular value is wrapped in an async container when assigned or passed.

Asynchronous Methods

In the above discussion of asynchronous storage, one kind of declaration that was not seen was the return value of a function. The reason is that a function result is logically not a storage location, i.e. a variable, it is a value. As was seen, returning a source is an effective way of returning a value asynchronously from a function.

However, the function itself may need to be asynchronous, avoiding all potential for blocking the thread of execution while waiting for results.

In Axum, there are few sources of blocking:

- Direct or indirect use of .NET synchronization primitives such as Monitor.Enter().
- Direct or indirect use of synchronous I/O, e.g. Console.ReadLine().
- receive expressions (discussed later)
- receive statement (discussed later)

Blocking is an unfortunate limitation, but a reality of programming .NET. Furthermore, it is quite cumbersome to program library-based solutions to avoid all blocking. Axum can automatically transform methods into asynchronous constructs, aided by the programmer. It does not, however, avoid blocking issues related to Monitor.Enter and other non-Axum synchronization primitives. These should not be used in Axum, which uses messages and empty/full variables for all data-exchange synchronization needs. Synchronization for protection purposes are handled declaratively.

Writing non-blocking code by hand is typically complex, tedious, and error-prone. The Axum compiler takes care of the tedious and subtle rules of doing it and allows you to concentrate on the logic of the algorithm itself.

It does so in methods and functions declared as ‘asynchronous.’ In such methods, receive expressions and statements are implemented without blocking the thread, and control-flow constructs are also transformed using transformations of continuation points.

Within an asynchronous method, transformations are made not just for receives, but also for any calls where there is an asynchronous alternative that adheres to the .NET Asynchronous Programming Model (APM). That model relies on splitting an operation XXX into a 'BeginXXX' and an 'EndXXX' call, one to start the operation, one to collect the results.

For example, System.IO.Stream has a method 'Read' defined on it which has an APM alternative. Code within an Axum asynchronous method referring to Stream.Read() would instead be using Stream.BeginRead /EndRead and avoid blocking the thread.

```
private asynchronous void ReadFile(string path)
{
    Stream stream = new Stream(...);

    int numRead = stream.Read(...);
```

```
        while (numRead > 0)
        {
            ...
            numRead = stream.Read(...);
        }
    }
```

Using asynchronous methods in your Axum code can significantly reduce the cost of message-passing and doing I/O and thus improve its scalability immensely[12].

Using asynchronous methods is often over-kill and has a performance penalty for small methods and functions which do not do I/O or messaging. Therefore, the rule is that methods are synchronous and have to be explicitly identified as asynchronous. The only methods that are asynchronous by default are agent constructors.

Most Axum constructs are available for use in both synchronous and asynchronous methods. A notable exception is 'interleave' (discussed later), which may only be used in asynchronous methods.

The rule of thumb around declaring asynchronous methods is as follows:

1. Any method containing a receive expression or statement, or an interleave should be asynchronous.
2. Any method calling an API that is known to have an APM variant should be asynchronous.
3. Any method calling an asynchronous method should be asynchronous.

[12] The author was able to observe 500,000 simultaneously blocked agents on his laptop without seeing the thread pool create any additional threads (there were 6 threads before they were started, 6 threads when they were all sitting blocked). He didn't try more than that...

Expressions and Statements

Axum adds functionality also to the contents of function bodies and control flow. There are a number of statements and expressions added to the standard C / Java / C# syntax.

Send

Send provides a value to a target that, depending on its specific semantics, will store or propagate the value in some way, as discussed previously.

The send expression has the following syntax:

```
target <-- source
```

where *target* is of type ITarget<T>and *source* of type T.

Example:

```
var x = new OrderedInteractionPoint<int>();
var chan = new CtStockQuote("http://...");

x <-- 17;
chan::GetQuote <-- "MSFT";
chan::GetQuote <-- "SUNW";

chan::Quote --> y;
```

In this example, the buffer ' x' is first sent the value 17 (for whatever purpose); then the stock tickers 'MSFT' and 'SUNW' are sent to the stock quote channel, after which the result is linked with the buffer 'y'.

In the case of sending to a channel-based port with a correlated reply port, send is easily combined with a corresponding receive expression. The send expression is not void, but instead produces a value of the type:

```
class RequestCorrelator<ReplyType>
{
  abstract ISource<ReplyType> Reply { get; }
}
```

Example:

```
channel ChA
{
  input int [] Add : int;
}

...
```

```
var repl = (x::Add <-- {10, 25, 4711}).Reply;
...
// Do something else while the addition happens
...
var result = receive(repl);
```

or, more synchronously:

```
var result = receive((x::Add <-- {10, 25, 4711}).Reply);
```

Note that the reply source instance is associated not with the channel but with the specific send operation – this enables correlation of send and receive operations.

Like any source, a reply can be hooked into networks:

```
var buf = ... // Some ITarget<int> instance
var repl = (x::Add <-- {10, 25, 4711}).Reply;
repl --> buf;
```

receive

The unary operator receive takes an ISource<T> operand and produces a value of type T. While the most general receive functionality is provided by the general receive statement, described later, the receive function is convenient in its simplicity:

```
ISource<int> a = ...

var b = receive(a);
```

Receiving a request of a request / reply port

Receiving a request from a port with an associated reply port and then sending a reply is straight-forward but somewhat more complex. To enable correlation, each request gets a unique reply port instance.

The type of the receive expression when using a port with a correlated reply is defined by a generic, runtime-provided, type *ReplyCorrelator<RequestType, ReplyType>*.

Given:

```
channel Channel<T,S>
{
    output T Add : S;
}

var x = new Channel<int,string>(...);
```

the type of 'receive(x.Add)' (in C# syntax) is:

```
class ReplyCorrelator<int,string>
{
```

```
    public int Value { get; }
    public ITarget<string> Reply { get; }
}
```

Thus, receiving and sending a correlated reply can be done easily:

```
var x = new Channel<int[], int>(...);
var inp = receive(x.Add);
inp.Reply <-- Array.fold (+) 0 inp.Value; // We need non-F# code
```

Note again that the reply target is associated not with the channel but with the specific receive expression.

Alternatively, networks can be used to process requests in a fully asynchronous way:

```
var x = new Channel<int[], int>(...);
x::Add --> (arr => arr.Value.Aggregate(( x,y) => x+y) )) --> inp.Reply;
```

By the way, in that sample we relied on the LINQ extension method 'Aggregate' to perform the addition. LINQ queries are supported by Axum with the same syntax as in C#.

Since no receive function is involved, the code proceeds past the network construction without waiting for data. This is the value of receive – it allows data to drive a workflow's continuation while networks are used for dataflow-based computations.

tryreceive

Like receive, tryreceive[13] may be used to retrieve a value from a source.

Its definition is

```
bool tryreceive( ISource<T> src, out T val );
```

and is used to receive incoming data opportunistically: "if the data is already available, I'll deal with it, otherwise I'll do something else."

```
ISource<int> a = …;
int x = 0;

if (tryreceive(a, out x))
    Console.WriteLine("received " + b.ToString());
else
    Console.WriteLine("nothing available");
```

interleave

The interleave statement specifies that each of its "children" statements should cooperatively execute in parallel. There are two forms: static and dynamic interleave.

[13] Not yet supported

The general form of the static statement, which controls a set number of nested statements, is this:

```
interleave { stmt1; stmt2; … stmtN; }
```

The dynamic interleaved statement is similar in structure to a loop, but there is no defined order in which it executes the various instances of its body. Its general form is:

```
interleave ( declaration in expression ) { ... }
```

This will create a dynamic number of interleaved children, one for every member of the collection. A separate binding of the declared variable is made available to each instance of the statement body; the interleave statement cannot use a variable declared outside the construct as the control variable. The instances of the body are started in an order undefined by the language.

No two of the parallel children may be executed at the same time, but any number of them may be blocked at once. In other words, if there are two children, and both of them are blocked in a 'receive' statement or expression, then either one of them may be allowed to execute once they are unblocked. However, as soon as one is executing, the other child must hold off until the executing child is once again blocked.

The interleave statement is not implying any guarantee of fairness scheduling its children: pre-emption is not assumed (or expected, for that matter).

Cancellation of a single branch of an interleave statement is a non-exceptional situation and has no other side-effect than early termination of the child statement. This is done by executing a 'continue' statement within the child branch.

The second form of cancellation, 'break,' immediately cancels all children and exits the interleave statement.

Interleave statements can only be used within asynchronous methods.

receive

The receive statement has many similarities (syntactically and semantically) with switch statements. Like a switch statement, a receive statement consists of a set of guarded branches, only one of which is chosen for execution. Unlike switches, receive statements evaluate all their guards at once and there is no default branch.

Each branch is implicitly terminated by the next branch – there is no possibility of falling into the next one; placing a break statement at the end of a guarded branch is optional and supported only for harmony with the C tradition.

Example:

```
receive
{
  from t into a:
    Console.Writeline("Received: " + a + " from timer");
```

```
      break;
    from buf where value > 17 into a:
      Console.Writeline("Received: " + a + " from buffer");
      break;
}
```

The receive statement above has two branches. The first guard will wait for a value to be available from the source 't', which if it arrives first is placed in 'a.' The second guard will wait for a value greater than 17 to be available from 'buf,' which is also placed in 'a' if it's the first to be available. Note that by using a timer as one of the guard sources, the receive statement can be made to support timeouts.

Within a guard filter, the identifier 'value' has a special meaning: it is used within the optional filter expression to refer to the yet-to-be-consumed value. It hides any other names of the same spelling within the guard's filter expression.

Axum allows each receive statement to arrange its branches according to priority[14]. The highest possible priority is '0'; the next lower is '1', and so on. The relative priority values have no meaning outside each individual receive statement, they only serve to order the branches within a single receive.

Assigned priorities must be positive 32-bit integers – they do not have to be constants or known at compile time. If any branch is organized under a priority regimen within a given receive statement, then all branches have to be. It is an error for a priority clause to not be the first statement within a receive statement block.

Example:

```
int x = 0;
int y = 0;

do
{
  receive
  {
  priority x:
    from buf1 into a:
      Console.Writeline("Received: " + a + " from buf1");
      break;
    from buf2 where (value > 17) into a:
      Console.Writeline("Received: " + a + " from buf2");
      break;
  priority y:
    from buf3 into a:
      Console.Writeline("Received: " + a + " from buf3");
      break;
    from buf4 into a:
      Console.Writeline("Received: " + a + " from buf4");
  }
} while (true);
```

[14] Priorities are not yet implemented

Due to their great potential to cause starvation, priorities should be used with care in any Axum program. As indicated, however, they *can* be used to mitigate the potential for starvation. In the sample above, modifying either x or y would control the risk of starvation. To the branches under the 'x' priority, simply add the following statements:

```
x = 1; y = 0;
```

and add corresponding statements to the 'y' priority branches.

The general form of the receive statement is:

```
receive
{
[priority expr1:]
   from expr2 [where expr3 ] [ into lvalue1 ]: ... break;
[priority expr4:]
   from expr5 [where expr6 ] [ into lvalue2 ]: ... break;
...
```

}

Repeating Guards

Repeating guards are used in conjunction with receive statements, but have semantics that are distinct enough to warrant separate coverage.

Consider this loop:

```
do
{
  receive
  {
    from buf1 into a:
      stmt-1(a);
      break;
    from buf2 into b:
      stmt-2(b);
      break;
  }
} while (true);
```

If stmt-1 takes a long time and cannot be parallelized, our code is potentially missing an opportunity to execute stmt-2 in parallel with stmt-1, particularly if the two don't have any kind of resource or memory contention issues. As one guard of the receive statement is chosen, the other is made obsolete and will not be considered. Based on the anti-starvation guarantee, the program will eventually execute stmt-2, but it will serially happen after stmt-1 is done and the receive statement is executed again.

What a repeating guard does is turn the receive statement into a dynamic "forker" of work. A receive statement with repeating guards will allow a new branch to be created for every match of the guard, as long as the entire

receive statement hasn't been finished. Branches of the receive statement with non-repeating guards will be executed exactly once per receive statement execution, while branches with repeating guards will be spawned for as long as no branch executes a 'break all' or 'final' statement.

This means that we don't need the loop as long as we make both our branches have repeating guards:

```
receive
{
  repeat from buf1 into a:
    stmt-1(a);
    break;
  repeat from buf2 into b:
    stmt-2(b);
    break;
}
```

Note the syntax – repeating guards are marked by the 'repeat' guard prefix. This receive statement will process values from buf1 and buf2 for all eternity[15], assuming that neither one execute a 'final' or 'break all' statement.

The following example will accept and process values from 'buf2' until we see a "Done" message on buf1:

```
receive
{
  repeat from buf2 into b:
    stmt-2(b);
    break;
  from buf1 $ (value == "Done"):
    Console.WriteLine("I think that's enough!");
    final;
}
```

Note that sometimes there is no reason to capture the received value into a variable; sometimes, just getting any value, such as from a timer, or from a special buffer, is enough.

[15] Metaphorically speaking… ☺

Schema[16]

Up to this point, we have not had much of a discussion of the data that travels over channels: we have used simple types and lists. It is possible to send more structured data over channels, too, but that requires a specific definition in order to behave in a way that makes them suitable for passing between domains.

The first requirement is that such data must be serializable, i.e. have a representation that allows it to be copied and sent across a network channel. Without this, there is no way to guarantee that a channel can be used for distributed applications. Not all .NET types can participate in serialization, so we need to define a subset of the .NET type system that supports serialization.

Second, we would like such types to support loose coupling of domains and agents. As far as payload types are concerned, it would be useful if there's a chance of decoupling the payload type definition from types used internally within domains. That allows each side of a channel to version their types independently of each other and potentially independently of the payload types.

Third, we would like the costs of passing data between domains to be amortized based on necessity, so that data-passing is cheap except when needed. One very costly approach is to copy all data when passed, as that guarantees that it is unique within each domain. Immutable data, in particular, does not require copying when passed between domains, but we need a concept of deep immutability of entire object graphs.

Axum schema types support these three needs:

1. Types defined as scemas are inherently serializable and support .NET serialization.

2. Schema types intrinsically support compiler-generated conversion to and from instances of homomorphous types, whether schemas or non-schemas. This allows each side of a communication channel to internally use distinct types while exchanging data via schema payloads.

3. Schemata are reference types. All schema fields are immutable and the only reference types used within schema types must also be schema types (certain .NET types, such as 'string,' which are known to be immutable, are also allowed). Value types used in schema must not contain any fields with references except to schema types. Schema types may also use lists in field definitions.

Schema types are thus immutable and can be passed by reference in local channel communication and by serialization in remote channel communication.

[16] This defines the new schema semantics. Currently, schema are not implemented as immutable types. The specification in this document is forward-looking.

Schema Definition

Schema types are defined as records where each field is prefixed with either the keyword 'required' or 'optional.' If any single field is denoted as required or optional, all fields have to be thus denoted and the type is a schema. Schemata may also contain functions and rules, which will be discussed later.

```
schema Schema_A
{
    required string Name;
    optional string Address = "N/A";
}
```

As shown, optional fields may define a default value, which is used if the field is not initialized during construction or found in a coercion or de-serialization operation. If the field definition does not specify a default value, the field type's default value will be used.

Instance Creation

Schema instances are created by explicit construction[17], coercion from a homomorphous instance, or by de-serialization:

```
var expr = << SOME NON-SCHEMA TYPE >>;

// Coercion
var s1 = coerce<Schema_A> non-schema-expr;

// Construction
var s2 = new Schema_A { Name = "Niklas", Address = "123 Main St." };

// Deserialization
var s3 = Schema_A.ReadObject( stream );
```

Once created, no field of a schema instance can be altered.

Coercion is supported between a schema type and any reference or value type that has the same "shape," loosely defined as having a trivial mapping from one type to the other. Intrinsically, schema coercion looks for fields that are named identically and with types that are also homomorphous. Many coercions fail only at runtime, as the exact shape of an object is only known at runtime.

Creation by construction is supported using the property-setter syntax. Each required filed must be used, while optional fields may be left out[18].

[17] Object graphs with cycles cannot be created this way. Coercion or de-serialization is required for such graphs.

Required and Optional Fields

The 'required' and 'optional' fields are primarily intended to allow some flexibility when coercing or de-serializing schema instances. A required schema field must have a mapping (explicit or implicit) in the source of de-serialization or the non-schema type involved in a coercion. An optional field does not have to have a mapping.

When creating a schema instance from a stream (de-serializing it) or coercion from an instance that does not have a given optional field, the field's default value is used instead.

Rules

When passing data between domains, it is important that each side of the communication can agree on what constitutes well-formed data and that any errors are caught at the source. This is expressed using schema rules, which are object invariants. It may be something as simple as not allowing empty strings:

```
schema Schema_A =
{
    required string Name;
    required string Address;

    rules
    {
      require !String.IsNullOrEmpty(Name);
    }
}
```

Rules form a special category of members: their use is entirely by the runtime and compiler; the compiler should therefore disallow any direct calls to rules from user-written code.

Each set of rules is checked every time rules are enforced.

All rules are applied and enforced after construction of an entire object graph, i.e. at the outermost level of the construction call chain. They are also enforced after de-serialization from disk or network I/O and when crossing a domain boundary.

If a rule can be shown to only depend on the data within what is reachable from the schema definition, the rule's result can be memoized. However, if a rule calls something like 'DateTime.Now[19],' which will obviously produce a new value each time it is used, the rule cannot be automatically memoized and must also be enforced at the point that the schema instance is sent outside a domain.

---

[18] The compiler actually generates a single constructor with all the required and optional fields as arguments. The constructor call syntax is then transformed into using this one call instead, passing in default values when an optional field is not found in the constructor call expression.

[19] If such rules seem odd, recognize that such a rule can be used for a simple implementation of time-to-live objects.

Functions

Schema types may define functions as long as they do not have any side-effects on the fields of the schema or anything reached from the schema reference. Among other things, functions are useful in expressing rules readably:

```
schema Schema_B
{
    required string Name;
    required string Address;
    optional DateTime CreationTime = DateTime.Now;

    private function TimeSpan TimeSinceCreation()
    {
        return DateTime.Now - CreationTime;
    }

    public function override ToString()
    {
        return String.Format{"{0}, {1}", Name, Address);
    }

    // This rule is re-evaluated every time the object crosses a
    // domain boundary.
    rules { TimeSinceCreation() < TimeSpan.FromHours(1); }
}
```

Inheritance

Schema types may inherit only from other schema types.

Channels Revisited

Channels were covered to some extend earlier in order to explain agents, but we need to revisit them and provide more detail about some of their elements.

Ports

Ports must be defined in terms of schema, value types containing only values or schema references, tuples, and lists. Certain .NET references types that are known to be immutable, such as 'string,' are also allowed as payload types.

Patterns

Each conversation must be terminated in the pre-defined state 'End.' It is a protocol violation to dispose a channel that is not at 'End' or at a state with an empty transition to 'End.'

In addition to the port-triggered transitions, patterns may use empty transitions. An empty transition is one which is triggered without consuming any input "token," and is taken when the target state has a matching trigger. It is particularly useful in defining a state as a potential end state:

```
channel Channel_A
{
  input  Pair<int, int> Add;
  output int Sum;

  Start: { Add -> S1; -> End }

  S1: { Sum -> Start; }
}
```

Functions

Functions are allowed in channel definitions and used to support pattern matching. For example, a condition may be used to further restrict schematized data rules:

```
channel HotelReservation
{
  input  Schema_A ReserveRoom;
  input  string ProvideCreditCard;
  output Pair<bool, string> Confirmation;

  function bool IsSuspicious(Schema_A x)
```

```
  {
    return "John Smith" && x.Address = "123 Main Street";
  }

  Start: {
    ReserveRoom $ IsSuspicious(value) -> Suspicious;
    ReserveRoom-> NotSuspicious;
  }

  Suspicious {
    ProvideCreditCard -> NotSuspicious;
  }

  state NotSuspicious =
  {
    Confirmation -> NotSuspicious;
  }
```

Channel functions are always protected members of the channel.

Mobile Channel Endpoints

In addition to value types, immutable types, and schema, channel endpoints are valid payloads of channel ports. This is an example of another form of safe inter-domain information exchange: ownership transfer. It means that a mutable object, after being sent away, is no longer usable by the sender.

What this means in practice is that a channel port can be used to send an endpoint of another channel, which generalizes the support for sub-patterns like request / reply. With a mobile channel end, any sub-pattern that can be expressed as a channel can be supported and correlated.

For example:

```
public channel Chan2
{
    input int Request;
    output int Reply;
}

public channel Chan3
{
    input Chan2.implements HereYouGo;
}

...
Chan3 ch3 = new Chan3("...");
Chan2 ch2 = new Chan2();

ch3::HereYouGo <-- ch2.Reversed;

ch2::Request <-- 10;
```

```
int x = receive(ch2::Reply);
```

As evident from that example, defining the port in Chan3 is one of the situations where the type name of the implementing side of a channel may be necessary to know explicitly. If we wanted to send the using end instead, it would look like:

```
public channel Chan2
{
    input int Request;
    output int Reply;
}

public channel Chan3
{
    input Chan2 HereYouGo;
}

...

Chan3 ch3 = new Chan3("...");
Chan2 ch2 = new Chan2();

var ch2i = ch2.Reversed;

ch3::HereYouGo <-- ch2;

int x = receive(ch2i::Request);
ch2i::Reply <-- x + 10;
```

When used in distributed scenarios, mobile channel ends may be transport-dependent. Not all transports (HTTP, TCP, etc.) will be able to support them, and not all network configurations will allow them to be used. Firewalls, for example, may place restrictions on what patterns are possible.

After sending a channel end, the local copy is no longer useable and any attempt to do so will result in an exception. Ownership of the endpoint has been transferred to the party receiving it. This characteristic of channel ends also affects the ability to route channel end points through broadcasting networks: only one of the copies will be valid.

For example, this code is invalid:

```
Chan2 ch2 = new Chan2();

var ch2i = ch2.Reversed;

ch3::HereYouGo <-- ch2;

ch2::Request <-- 13;    // Error: ownership of the channel end is gone
```

and so is this code on the other side of ch3:

```
ch3i::HereYouGo -<< { f1, f2, f3 } >>- buf1;
```

Inheritance

One channel may inherit from another channel while maintaining the “is-a” property. Channel inheritance affects both the ports and the pattern. For ports, the derived channel simply extends the set of ports available for communication: there is no shadowing of ports. Channels may not inherit types that are not channels.

Abstract channels are also possible: states and functions may be left undefined, to be overridden in a concrete channel derived from it.

```
channel StartupTermination
{
  input StartupData Startup;
  input bool Done;

  Start:
  {
    Startup $ (value.Name == null) -> BadData;
    Startup -> UpAndRunning;
  }

  BadData: {}
  UpAndRunning: {}

  Terminating:
  {
    Done $ value -> End;
    Done -> UpAndRunning;
  }
```

Abstract state definitions are primarily useful when defining some base protocol for startup and shutdown, leaving the details of the interim states to derived channels:

```
channel Purchasing : StartupTermination
{
  input PurchaseOrder PurchaseStuff;

  UpAndRunning:
  {
    PurchaseStuff $ (value.ItemCount > 0) -> Terminating;
    -> BadData;
  }

  BadData:
  {
    ... details left to reader’s imagination ...
  }
```

If any state is left undefined in a derivation from an abstract channel, the derived channel is abstract.

Patterns and Inheritance

Outside of the case with abstract pattern states, the rules for patterns involved in channel inheritance are more complex: the basic rule is that, from the using (client) perspective, there must not be any transitions taken or new ports used within the pattern, unless initiated from the using side of the communication. In other words, if the side implementing the channel type is using the derived channel, and the client is using the base, the conversation will still be possible. This retains the 'is-a' relationship from the using perspective. The derived channel is said to be "using-perspective compliant" with the original channel. Axum does not support a notion of implementing-perspective compliance for derived channels.

To comply, the pattern of a derived channel may do the following[20]:

- Add new states.
- Add new transitions to existing states, as long as these transitions do not introduce any ambiguity into the pattern, and as long as the prefix ports of any new transition are all input ports.

Functions defined in any channel declaration are implicitly 'virtual,' i.e. redefinition in a derived network overrides the base implementation of the function.

Non-compliance when deriving a channel is an error in the channel specification.

Empty

The Axum runtime contains a definition of an empty channel, which can be used to start agents with which there is no further interaction. It has the following definition:

```
channel Empty
{
  Start: { -> End; }
}
```

'Empty' is not guaranteed to work for connections over distributed transport protocols, which may require that a message be sent before establishing a connection and starting the server-side agent. When used with the default in-process "transport," the agent is guaranteed to be created and scheduled for execution.

[20] We also need a rigorous definition for how to add conditions to transitions in derived patterns.