Singularity Design Note

17

Introduction to Sing#

Language extensions for Singularity

The Sing# language extends the Spec#/C# family with special support for Singularity's abstractions such as message passing and channel contracts.

Table of Contents

1. Brief overview

This section informally presents the main features of the Sing# extensions via examples. Later sections provide reference style descriptions of all the features.

1.1. Channel Contracts

Channel contracts are central to Singularity's isolation architecture. Contracts are supported directly in Sing#. Here's a contract describing a simple interaction on a channel.

```
contract C1 {
  in message Request(int x) requires x>0;
  out message Reply(int y);
  out message Error();

  state Start: Request? -> (Reply! or Error!) -> Start;
}
```

The declaration of contract C1 also declares three messages: `Request` , `Reply`, and `Error`. Each message declaration specifies the types of arguments contained in the message. For example, `Request` and `Reply` both contain a single integer value, whereas `Error` does not carry any values. Additionally, each message may specify Spec# `requires` clauses restricting the arguments further.

Messages can also be tagged with a direction. The contract is always written from the exporter point of view. Thus, in the example, `Request` is a message that can be sent by the importer to the exporter, whereas `Reply` and `Error` are sent from the exporter to the importer. Without a qualifier, messages can travel in both directions.

After the message declarations, a contract specifies the allowable message interactions via a state machine driven by send and receive actions. The first state declared is considered the initial state of the interaction. In our example contract `C1`, we only declare a single state called `Start`. After the state name, action `Request?` indicates that in the `Start` state, the export side of the channel is willing to receive (?) a `Request` message. Following that the construct (Reply! or Error!) specifies that the exporter sends (!) either a `Reply` or an `Error` message. The last part `-> Start` specifies that the interaction then continues as in the `Start` state, thereby continuing ad-infinitum.

Even though the contract only specifies a single state by name, there is in fact an intermediate state in this interaction. The state after the first message is received is implicit in the protocol. If we wanted to, we could declare this state explicitly by writing the contract as follows:

```
state Start: Request? -> AfterRequest;

state AfterRequest: one {
   Reply! -> Start;
   Error! -> Start;
}
```

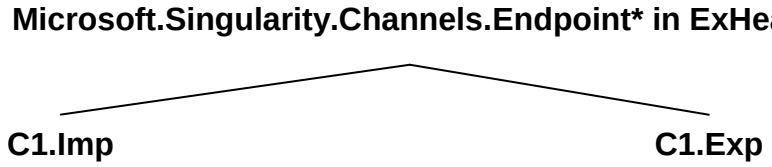
Here we name the intermediate state `AfterRequest`.

1.2. Endpoints

Channels in Singularity manifest as a pair of endpoints: the importing and exporting sides of the channel. Each endpoint has a type that specifies what contract the channel adheres to. These endpoint types are implicitly declared within each contract. A contract C1 is represented as a class, and the endpoint types are nested types within that class as follows:

```
C1.Imp // type of import endpoints of channels with contract C1
C1.Exp // type of export endpoints of channels with contract C1
```

Such contract specific endpoints derive from a common abstract endpoint as shown in the following hierarchy.



1.2.1. Send/receive methods

Each contract specific endpoint contains methods for sending and receiving the messages declared in the contract. In our example, we have the following implicit methods defined:

```
C1.Imp {
  void SendRequest(int x);
  void RecvReply(out int y) ;
  void RecvError();
}

C1.Exp {
  void SendReply(int y);
  void SendError();
  void RecvRequest(out int x)
}
```

Note how the importing endpoint has methods to send the *in* messages of the contract and to receive the *out* messages of the contract, whereas the exporting endpoint has methods to receive the *in* messages and to send the *out* messages. The semantics of the Send methods are that they send the message asynchronously. The receive methods block until the given message arrives. If a different message arrives first, an error occurs. Such errors should never occur if the program passes the contract verification check. Unless a receiver knows exactly what message it requires next, the above receive methods are not convenient. Instead, Sing# provides the **switch receive** statement.

1.3. Switch-Receive statement

Consider the following code which waits for either the Reply or Error message on an imported endpoint of type C1.Imp.

```
void M( C1.Imp a)
{
  switch receive
  {
    case a.Reply(x):
        Console.WriteLine("Got reply {0}", x);
        break;
    case a.Error():
        Console.WriteLine("Got error");
        break;
  }
}
```

The switch receive is a combination of two steps:

1. Blocking for a particular set of messages to be present on a set of endpoints

2. Receiving the set of messages and binding their arguments into local variables.

In the example above, the switch receive has two patterns, either receive Reply on endpoint imp, or Error on the same endpoint. In the first case, the integer argument of the Reply message is automatically bound to the local variable x, whose scope is the branch of the case it appears in.

The switch receive construct is however much more general, since patterns can involve multiple endpoints as shown below. In this example, we have two endpoints a and b on which we can receive reply or error messages:

```
void M( C1.Imp a, C1.Imp b)
{
  switch receive
  {
    case a.Reply(x) && b.Reply(y):
        Console.WriteLine("Got both replies {0} and {1}", x, y);
        break;
    case a.Error():
        Console.WriteLine("Got error reply on a");
        break;
    case b.Error():
        Console.WriteLine("Got error reply on b");
        break;
    case a.ChannelClosed():
        Console.WriteLine("Channel a is now closed");
        break;
  }
}
```

The example illustrates that we can wait for particular combinations of messages using the switch receive statement. The first branch is only taken if the Reply message is received on both endpoints a and b. The last case contains the patter **ChannelClosed**(), which is a special pattern that fires when the channel is closed (by the other party) and no more messages remain to be received.

1.4. Endpoint ownership

Endpoints are precious resources that need to be tracked at compile time in order to perform contract verification. To simplify this tracking, there is an ownership model associated with endpoints. Each endpoint is owned by at most one thread at any point in time, or by at most one data structure. E.g., if an endpoint is sent in a message from thread T1 to thread T2, then ownership of the endpoint changes from T1, to the message, and then to T2 upon receipt by T2.

To simplify the tracking of endpoints for the static analysis, endpoints can only be held directly in messages and local variables. This is obviously restrictive and the language provides a way to store endpoints indirectly within data structures through an abstraction called a TRef..

1.5. TRefs

A TRef is a storage cell of type TRef<T> holding an endpoint of type T. TRef has the following signature.

```
class TRef<T> where T:ITracked {

  public TRef([Claims] T i_obj);

  public T Acquire();

  public void Release([Claims] T newObj);
}
```

When creating a TRef<T>, the constructor requires an object of type T as argument. The caller must have ownership of the object at the construction site (as indicated by the [Claims] annotation on the parameter). After the construction, ownership has been passed to the newly allocated TRef. The Acquire method is used to obtain the contents of a TRef. If the TRef is full, it returns its contents and transfers ownership of it to the caller of Acquire. Afterwards, the TRef is said to be empty. Release transfers ownership of a T object from the caller to the TRef. Afterwards, the TRef is full.

TRefs are thread-safe and will block until full. The ITracked interface is used for call backs. E.g., the implementation of Endpoints uses these to associate the endpoint with the correct thread whenever the endpoint changes ownership.

TRefs live in the GC heap and contain a finalizer. If a TRef becomes unreachable when it is full, the finalizer will dispose of the endpoint contained within. As a result, the channel gets properly closed.

1.6. The Exchange Heap

Since ownership of blocks is transferred from one thread/process to another on message exchanges, we need a way to allocate and track blocks of memory that can be exchanged in this fashion. The channel system requires that message arguments are either scalars, or blocks in the exchange heap. There are two kinds of blocks in the exchange heap: individual blocks, or vectors. Their types are written as follows:

```
using Microsoft.Singularity.Channels;

…
R* in ExHeap pr;
R[] in ExHeap pv;
```

The type of pointer pr specifies that it points to an R struct in the exchange heap. ExHeap is actually a type defined by the runtime system that provides allocation, deallocation and other support for this heap. The type of pv is a vector of Rs in the exchange heap. The fully qualified name is Microsoft.Singularity.Channels.ExHeap. The type of R has to be either a primitive value type (int, char, etc.), an enum, or a rep struct. Rep structs are explained in more detail in section 2.3.

1.6.1. Allocation and de-allocation

Here's an example of how to allocate blocks in the exchange heap and how to free them.

```
rep struct R {
  public int x;
  public int y;

  public R(int a, int b) { this.x = a; this.y = b; }
}

void M() {
  R* in ExHeap pr = new[ExHeap] R(1,2);
  int[] in ExHeap pv = new[ExHeap] int[20];

  pr->x = pr->x + pr->y;

  pv[5] = pr->x;

  …
  delete pv;
  delete pr;
}
```

Indexing of vectors in the exchange heap is safe through the use of runtime bounds checking. As the example makes clear, blocks in the exchange heap are explicitly managed, i.e., they have to be explicitly de-allocated. The ownership checker of Sing# makes helps getting that correct.

1.6.2. Ownership tracking

The compiler tracks ownership of resources such as blocks in the exchange heap (including endpoints that are implicitly allocated in the exchange heap), as well as special objects in the GC heap with interface ITracked such as ESets and EMaps (see section 2.2.3). We refer to these as tracked types.

The rules can be summarized as follows. At any point in time, a single stack frame owns a particular tracked resource. Operations that affect the ownership are:

- new: on completion, the allocated resource is owned by the current stack frame
- delete e: the expression e must evaluate to a pointer or vector in the exchange heap and the current stack frame must own it. After the operation, the current stack frame no longer owns the resource.
- e.Dispose(): the expression e must evaluate to a GC object with the ITracked interface. The current stack frame must own the object prior to the call and does no longer own it subsequently.
- return: if a method return type is of tracked type, then the callee must own the resource it returns at the return point. The caller assumes ownership of the result of the call.
- parameter:
    - an unannotated parameter of tracked type is said to be *borrowed* by the called method, i.e., ownership is temporarily transferred from caller to callee, but reverts to the caller upon return. The callee cannot retain or dispose of the passed resource.
    - a parameter of tracked type annotated with **[Claims]** specifies that ownership of the passed resource transfers permanently from caller to callee.
- By-ref parameter: on entry, the caller passes ownership permanently of the resource contained in the ref cell on entry to the callee. On return, the callee passes ownership of the resource contained in the cell back to the caller. This can be satisfied in 2 ways. Either the cell contains the same resource and

ownership of the resource was not consumed by the callee, or the cell contains a new resource that the callee provides and the callee consumes the original resource in the cell.

- Out-parameter: on exit, the cell must contain a resource owned by the callee at the return point and its ownership transfers to the caller.

2. Reference

2.1. Contract syntax and semantics

Contracts are written from the viewpoint of the exporting party. Thus, if a contract states that it receives a message M at a certain point, then the exporting party must receive this message, and the importing party can send that message.

2.1.1. Basic contracts

A contract consists of message type declarations, as well as state declarations. Contracts are named and possibly generic over types (or other contracts). Syntactically, contracts have the following general form:

```
contract Id [BoundVars]  {
    MessageDeclarations

    StateDeclaration
}
```

Each message declaration specifies an optional direction, a message name and a sequence of message argument types. The direction is either **in** or **out**, limiting the use of the message to be either received or sent by the exporting side. Without a directional qualifier, the message can be used in both directions. The bound variables allow one to parameterize the message by types or contracts.

```
[in | out] message MessageId [BoundVars]( Type [Id], …, Type  [Id] );

BoundVars ::= < Id, …, Id >
```

2.1.2. Contract states

[NOTE: Currently, Sing# only allows the "one" form of states and does not permit mixed send/receive alternatives]

A state declaration has the following general form:

```
state StateId [BoundVars]: ( one | all ) {
   MessageSequence1 ;
          …
   MessageSequencen ;
}
```

The state identifier names the state. The state kind states whether the list of message sequences should be interpreted as a choice of **one** message sequence listed, or an arbitrary interleaving of **all** message sequences listed. In the case of a choice, message receives are interpreted as a choice made by the importer of the contract (the client), message sends are interpreted as a choice made by the exporter (the server).

If there are both receive and send choices present, we restrict the forms to two cases:

1. All receives are listed first, all sends are listed second.
   In this case, the choice of doing a send is with the importing side of the contract. The exporter must be willing to receive the listed receive messages, and can only send a message if the queue is either empty, or contains a message not listed among the receive messages.
   There is still a race in such a contract that needs to be resolved in the continuations of the message sequences starting with a send.

2. All sends are listed first, all receives are listed second.
   In this case, the choice of doing a send is with the exporting side of the contract. The importer must be willing to receive the listed send messages, and can only send a message if the queue is either empty, or contains a message not listed among the send messages.
   There is still a race in such a contract that needs to be resolved in the continuations of the message sequences starting with a receive.

In the case of **all** interleavings of messages sequences, the choice of order lies with senders of messages. For example, the snippet **all** { M! -> A; N! -> B; } would allow the sender to send message M and N in any order and the receiver has to be ready to receive them in any order. Similarly, if the interleaving specifies receives all { M?-> A; N? -> B; }, the choice of order lies with the sender. The receiver is willing to receive them in any order. For mixed interleavings of sends and receives, the same reasoning applies: the sender gets to choose the order.

Message sequences start with an action, followed by a continuation.

```
MessageSequence ::= Action Continuation
```

Actions are either sends **!** or receives **?** of messages declared in contracts. The send or receive symbols are always required, even if the message is declared with a direction in order to improve readability.

```
Action ::= MessageId [BoundVars]!
       |   MessageId [BoundVars]?
```

Continuations are either empty (ending the message sequence) or consist of further actions, next states, or inline choices:

```
Continuation::= /* epsilon */
            |   -> StateId [BoundVars] Continuation
            |   -> Action Continuation
            |   -> ( ActionSequence or … or ActionSequence ) Continuation

ActionSequence::= Action ( -> Action)*
```

The action sequences in an inline choice must start with the same kind (all sends or all receives). Final states in a contract are those containing no message sequences.

2.1.3. Subroutine states

To reuse descriptions of finite message exchanges at several points in a contract, we allow a continuation to use a state as a sub routine. A state mentioned in a message sequence is a subroutine state, if it isn't followed by a semicolon. Here's an example contract using a subroutine state.


```
  state Start:  Process -> Start;

  state Process: one {
    M1? -> (A1! or A2!)
    M2? -> (A3! or A4!)
  }
```

In the Start state, we first accept message sequences according to subroutine state “Process”, then the protocol repeats to the Start state. The syntax of the Start state uses the short hand introduced in section 2.1.6.
A message sequence with a subroutine state evolves according to the named state. When all messages exchanges are exhausted according to that state, the contract evolves according to the remaining continuation. We require the named subroutine state to describe message exchanges of only finite length. Contracts not satisfying this restriction are considered ill-formed.

2.1.4. Joins

The semantics of subroutine states provide a way to express joins in the contracts. A join is like a barrier, waiting for a set of possible traces to end before resuming the continuation of the contract. If a subroutine state contains parallel message sequences, then the continuation after the subroutine state, starts only after those message sequences have all been exhausted.

2.1.5. Endpoint states

When referring to an endpoint in types such as C.Imp or C.Exp, one has the ability to specify the channel state explicitly:

```
TypeRef::= ...
           |  TypeRef : Id
```

The states of a contract are turned into nested types of the contract and are referenced thus just like ordinary types. The form C.Exp:S specifies that the endpoint is in state S of contract C. Without the colon qualifier, endpoints are assumed to be in the start state. Note however, that these states are only required at abstraction boundaries, e.g., on method parameters and results, and on TRef instances. Local variables never need such state annotations, since the state is inferred locally.

2.1.6. Shorthands

States consisting of a single message sequence can be written without kind and braces, and the message sequence doesn’t have to start with an action.

```
state StateId [BoundVars]:  Action Continuation ;
state StateId [BoundVars]:  StateId [BoundVars] Continuation ;
```

2.1.7. Contract extensions

Contracts can be defined as extensions of existing contracts. Contract extensions are non-symmetric with respect to the importing and exporting sides of a channel. The exporting side of a contract extension is compatible with the exporting side of the base contract. The importing side of the extension is unrelated to the importing side of the base contract.
A contract extension can define new messages, new states, and extend existing states. The message sequences in new states are unrestricted. An existing state is extended by using the **override** keyword.


```
contract C {
   message A();
   message B();

   state Start: one {
     A! -> End;
   }

   state End: ;
}

contract D : C {
   message E();
   message F();
   message G();

   override state Start: one {
     E! -> Other;
   }

   state Other: one {
     F? -> G! -> Other;
   }
}
```

Contract D extends contract C by adding 3 new messages (E, F, G) and a new state (Other). The existing Start state is extended by adding a new message sequence guarded by the send of the new message E. The following rules apply to state overriding.

Rule 1: The overridden state in the base contract must be a choice (one) state.

Rule 2: Every message sequence in the overridden state in the base contract starts with a send.

Rule 3: The overriding state must be a choice (one) state.

Rule 4: Every message sequence in the overriding state starts with a send of a new message (declared in the contract extension).

Rule 5: Upcasts must respect the protocol state. Thus, an upcast is only possible if the state of the endpoint is an overridden state present in the base contract that is being cast to.

Rule 6: Downcasts must respect the protocol state. Downcasts don't change the protocol state.

The rational for these rules is that the channel interaction must occur at a contract level agreed upon by both the importing and exporting sides. This common level is defined by the view of the exporting side and the fact that the importing side cannot drive the protocol from a base state to a state of the derived contract. Transition from a base state to a derived contract state is performed by a send by the exporting side, which thereby signals to the importing side which level it is serving.
The common usage scenario for sub contracts is that there is some common base contract C for a set of services. But each service may add additional functionality in the form of a subcontract. Clients connect to services via a general mechanism that expects a C.Exp to be connected to the service. The service may downcast the endpoint to see if it can serve a more refined service and if so, reply with a message (such as E in the example above) to transition the protocol into a state of the derived contract that enables the additional functionality.

2.2. Channel Related Types

2.2.1. Message tags

Each contract C contains an enumeration C.Tags with elements corresponding to each declared message in the contract. These are used for implementation purposes and the Sing# user doesn't usually need to interact with message tags explicitly.

2.2.2. Endpoints

For each contract C, there are implicit nested types C.Exp and C.Imp representing the two types of endpoints of a channel adhering to contract C. Endpoints contain methods for sending and receiving particular messages. For each message M(T_1 x_1, … T_n x_n), there are methods:

```
public void SendM(T1 x1, … Tn xn);
public void RecvM(out T1 x1, … out Tn xn);
```

SendM is non-blocking, and RecvM is blocking. RecvM blocks until a message is available and fails if a message other than M is at the head of the queue.
The receive methods can be used when it is known what the next message type is a-priori. Otherwise, a *switch receive* statement is more appropriate.

2.2.3. Endpoint Sets

If a thread wants to handle an unbounded number of endpoints of a certain type, it can use an endpoint set. Endpoint sets support two basic operations: 1) an endpoint can be added to the set explicitly, via the Add method, and 2) switch receive can select a message from any endpoint in a given set. See the switch receive section 2.4.1 below. The signature of endpoint sets is as follows.

```
public class ESet<EP> where EP : Endpoint {
  public void Add([Claims] EP endpoint);
}
```

The endpoint type parameter must be instantiated to the particular contract and Imp or Exp endpoint type as well as the state the endpoint is in, e.g., ESet< MyContract.Imp:Ready >.

2.2.4. Endpoint Maps

Endpoint maps are similar to endpoint sets, but in addition to holding endpoints, the map associates with each endpoint an arbitrary data item that is recovered when a message is received on such an endpoint. See section 2.4.1 for more details. The signature of endpoint maps is as follows.

```
public class EMap<EP, D> where EP : Endpoint {
  public void Add([Claims] EP endpoint, D data);
}
```

In addition to the endpoint type, EMaps are also instantiated with the type of data that is held for each endpoint. E.g., EMap< MyContract.Imp:Ready, int > associates an integer with each endpoint in the set.

2.2.5. Contracts

Each contract C is represented as a type. There are never instances of such contract types, but they encapsulate a static method for creating new channels of a given contract:

```
public static void NewChannel(out C.Imp! imp, out C.Exp! exp);
```

The `NewChannel` method returns the two endpoints of the new channel. The channel starts out in the start state of the channel (the first listed state). The ! type suffix states that the resulting values are non-null. The non-null types are a general feature inherited from Spec#. The typical creation sequence for a channel is as follows:


```
C.Imp! imp;
C.Exp! exp;
C.NewChannel(out imp, out exp);
```

There exists another overload for `NewChannel` that also takes the initial channel state as an argument.

```
public static void NewChannel(out C.Imp! imp, out C.Exp! exp, int initialState);
```

The typical usage for creating a channel in state `S` of a contract `C` is:

```
C.Imp! imp;
C.Exp! exp;
C.NewChannel(out imp, out exp, C.S.Value);
```

2.2.6. States

Each state S of a contract C is turned into a nested class S.C with abstract methods encoding the state transitions. These methods only serve as an encoding of the contract for the model checker. The only useful aspect of states for the programmer is their use in specifying what state a particular endpoint is in on parameters, results, and on TRef, ESet and EMap instantiations. The following syntax is used in each of these cases:

```
C.Imp:Ready Foo(D.Exp:Start exp) { … }

TRef<C.Imp:Ready>
ESet<C.Imp:Ready>
EMap<C.Imp:Ready>
```

Method `Foo` states that the argument endpoint exp must be in the `Start` state, and that the result endpoint is in the `Ready` state. The instantiations state that the associated endpoints are in the `Ready` state.

Another useful feature of states is to use them in runtime checking to determine what state a particular endpoint is in. The following method has a precondition that requires the endpoint to be in state `Ready` or `Waiting`.

```
void M( D.Exp exp)
  requires exp.InState(D.Ready.Value) || exp.InState(D.Waiting.Value);
{ .. }
```

The `InState` method takes an integer argument that encodes the state of the endpoint. Each state in a contract has a constant associated with it called `Value`.

2.3. Rep types

Rep types are structs that are suitable for allocating in the exchange heap. The restriction on such structs is that their fields have types that are themselves exchangeable over channels, such as primitive scalars, enums, or pointers into ExHeap. Rep types cannot contain endpoints however.

Rep types are declared like structs, but with the rep prefix. The following is an example of a rep struct with two integer fields and a pointer to a char vector in the exchange heap.


```
using Microsoft.SingSharp;

public rep struct NamedPoint : ITracked {
  public int x;
  public int y;
  public char[] in ExHeap name;
}
```

The ITracked marker interface is required here because the struct contains objects that need to be tracked themselves, in this case the char[] name.

2.3.1. Pointer-free structs

Network header parsing is made easier if we allow a sequence of bytes to be interpreted as a particular structure. In order to support this safely, Sing# introduces the notion of a *pointer-free struct*. Fields of pointer-free structs contain only fields with pointer-free types. Pointer-free types are all the primitive numeric types as well as Booleans and characters, as well as pointer-free structs.

Pointer-free structs are declared like ordinary structs but with the qualifier pointerfree. They form a special sub-category of rep types, but they are useful irrespective of channels.

```
pointerfree struct S {
   int x;
   float y;
   R r;
}

pointerfree struct R {
   int u;
   char c;
}
```

Structs S and R are both valid pointer-free structs. Note that S is allowed to contain a struct field of type R, since R is declared to be pointer-free itself. Pointer-free structs can be used wherever ordinary structs are acceptable. But their utility comes in conjunction with overlays (Section 2.4.2).

2.3.2. Struct inheritance

Rep structs can inherit from other rep structs. The fields of a parent rep struct are visible to the sub struct under the same rules as for classes. It is possible to call methods in base structs using the **base** syntax. Virtual methods can be defined as well. However, these are only useful when used in conjunction with interfaces. Struct inheritance allows one to safely cast a pointer S* **in** ExHeap to T* **in** ExHeap, provided that struct S inherits from T. Similarly, one can pass by reference a struct S where a **ref** T parameter is expected. There is no conversion from S to T on values themselves.

2.4. Statements

2.4.1. Switch receive

In order to wait for messages or message patterns on a variety of channels, Sing# supports a switch receive statement which is commonly known as select-receive. We chose the alternative keywords in order to avoid making the common word “select” a new statement level keyword. The syntax is:


```
  Statement ::= ...
              |  switch receive { ReceiveCase* }

  ReceiveCase ::= case ReceiveConjunct ( && ReceiveConjunct )* [ in Id [ ~> Id ]]:
CaseBlock
                |  case timeout: CaseBlock
                |  case unsatisfiable: CaseBlock

  ReceiveConjunct ::= Id . M ( [Type1] Id1 … [Typen] Idn )
```

The normal receive case consists of message receive conjuncts, where each receive conjunct consists of a local variable Id referring to a non-null endpoint, a message M, and a sequence of variables that are bound to the arguments of the message in the corresponding CaseBlock. The optional type declarations preceding the message argument variables permit the programmer to repeat the declared message argument types.

Within a ReceiveCase, the endpoint identifiers among the conjuncts must be distinct, ie., it is not allowed to receive two messages on the same channel in a single ReceiveCase. The semantics of a switch receive is that the thread blocks until one of the cases is enabled. A case consisting of receive conjuncts is enabled when all the conjuncts are satisfied. If so, the messages are received and their arguments bound to the variable identifiers, which are then available in the CaseBlock.

If present, the special case **timeout** fires when no message receive conjuncts are enabled. Currently, the timeout span is 0ms, but we should support non-zero timeouts in the future. The **unsatisfiable** case is only enabled on an error condition, i.e., when no other case can ever be taken (e.g., when all queues contain a message, but no pattern is statisfied).

The optional suffix **in** followed by an endpoint set makes the last ReceiveConjunct into a receive from an endpoint set (ESet). In that case, the endpoint Id of the ReceiveConjunct acts as a pattern variable and is bound to the actual endpoint in the set from which the message was received. Note that if the branch is taken, the endpoint is removed from the set.

The extra suffix **~>** followed by a variable pattern is used on endpoint maps (EMap) where the variable binds the data associated with the endpoint on which the message was received. Again, the endpoint is removed from the map if the branch is taken.

2.4.1.1. Channel closed pattern

The special message `ChannelClosed` can be used to recognize if a channel has been closed by its peer. It is only satisfied if the channel is closed and no more messages are outstanding. For example:

```
  switch receive
  {
    case a.ChannelClosed():
        Console.WriteLine("Channel a is now closed");
        break;
  }
```

2.4.2. Overlays

In order to facilitate parsing and constructing network packet headers, Sing# allows viewing a sequence of bytes (or any sequence of pointer-free data) as a pointer-free structure. For this purpose, a new kind of local declaration is introduced:

```
  LocalDecl ::= ...
              |  ref TypeReference Id = ref Expression [ Expression ] ;
```

`TyperReference` must refer to a pointerf-free struct S. The declaration introduces `Id` as a new local variable of type S & (or managed reference to S). In C, the equivalent statement would be


```
S* s = (S*)&e1[e2];
```

The compiler checks that `e1` is an array of pointer-free elements and `e2` is a numerical index. At runtime, the code first checks that `e1[e2]` is a valid array element and then checks that array `e1` is large enough to hold a structure S at index `e1` without overflowing the array. The exact check performed is

```
if ( (e1.Length – e2) * sizeof(e1.ElementType) < sizeof(S) ) {
   throw new ArgumentOutOfRangeException();
}
```

Once a new struct reference has been initialized, it can be used in the same way as a reference parameter would be used. Reading and writing parts of the entire struct reads and writes the corresponding bytes of the underlying array. Here's an example:

```
pointerfree struct S {
  int x;
  int y;
}

int Test( int[] a, int index) {
   ref S s = ref a[index];
   return s.y;
}

static void Main() {
   int[] a = new int[]{0,1,2,3,4,5,6,7};
   assert( Test(a, 5) == 6 );
}
```

2.5. Ownership tracking

The Sing# compiler tracks ownership of resources that are allocated explicitly, such as channel endpoints, rep structs in the exchangeable heap, as well as data structures such as ESets and EMaps that contains such resources. The compiler enforces statically that tracked resources are

1. Properly deleted when no longer used
2. Not used when deleted or ownership has been delegated,
3. Ownership delegation is correctly implemented according to the parameter/return specifications

We refer to objects of the above types as tracked types. A type is tracked if either a) it derives from the ITracked interface, or b) it is allocated in an explicitly managed heap, such as the exchange heap `Microsoft.Singularity.Channels.ExHeap`.

2.5.1. Fields of tracked types

Access to fields of tracked structs or tracked objects is governed by the type of the field. There are two cases. If the field type is non-tracked, then reading and writing the field is unrestricted. If the field type is itself tracked, then access to the field is only permitted in contexts where the enclosing struct/object is *exposed*. An object/struct pointer can be exposed using an expose block.


```
    expose (e)
    {
       … access to tracked fields of e granted here
    }
```

The mental model for expose blocks is that the ownership invariant of the target of the expose holds on entry and exit of the block. Within the block the ownership invariants of the fields can be temporarily violated and the compiler makes sure that by the end of the block, they are reestablished.

The following example deals with a linear list structure List that contains an integer and a next field to the tail of the list. SplitTail

```
ListElem* in ExHeap SplitTail(ListElem* in ExHeap list)
{
  expose (list)
  {
    ListElem* in ExHeap result = list.next;
    List.next = null;
    return result;
  }
}


rep struct ListElem : ITracked {
  int x;
  ListElem* in ExHeap next;
}
```

2.6. Non-null types

Whether a pointer may be null or is definitely not null is a very common specification. To make it easy to specify that pointers are non-null, Spec# and Sing# provide non-null types. For any reference type T, the type T! denotes a reference to a T that is definitely non-null. Such non-null types can be used wherever types occur with a few exceptions.
The type checker ensures that when a context expects a non-null value, the program actually supplies one. For example, when passing a possibly null value as the argument to a method whose parameter has a non-null type, the checker will complain.

2.6.1. Fields of non-null type

When non-null types are used for fields, they act as object invariants. Reading such fields should always provide a non-null value and writing such fields can only be done with non-null values. In order to make such guarantees, the checker must ensure that all non-null declared fields of an object are initialized before any code could read them and observe the null value from the default initialization.

The compiler enforces the initialization of fields in one of two ways, depending on whether the constructor is *delayed* or *non-delayed*.

2.6.1.1. Delayed constructors

A delayed constructor treats **this** as a *delayed reference*, meaning that the constructor is only allowed to initialize the object pointed to by **this**, but does not actually read fields of the object. We call the **this** parameter of a delayed method a *delayed reference*. Delayed references can only be used to initialize fields of the delayed reference, or they can be stored into other objects that are themselves delayed. If a delayed reference needs to be passed to another method (as a explicit parameter or as **this**), then the method signature must guarantee

that it treats the parameter as delayed by having the [Microsoft.Contracts.Delayed] attribute on the parameter (method for **this**). Thus, base constructors called from a delayed constructor must themselves be delayed.

By default, the compiler assumes constructors are delayed. To override this default, add the attribute [Microsoft.Contracts.NotDelayed] on the constructor. This is useful if the constructor does non-trivial computations using the fields of the object being initialized. Here's an example of a delayed constructor.

```
class C {
  string! name;
  // A delayed constructor (default)
  public C(string! n) {
     this.name = n;
  }
}
```

For delayed constructors, the compiler enforces that all non-null fields are initialized by the end of the constructor. This is sufficient, since the base constructor called (implicitly or explicitly) by a delayed constructor must also be delayed, thereby guaranteeing that fields will not be read (via virtual calls). Most constructors in programs are delayed. This property allows one to guarantee proper initialization of circular structures, meaning that all pointers on a cycle are actually non-null as in the next example:

```
class List {
  Node! head;
  public List() {
    this.head = new Node(this);
  }
}
class Node {
  List! list;
  Node! prev;
  Node! next;
  public Node([Delayed] List! list) {
    this.list = list;
    this.prev = this;
    this.next = this;
  }
}
```

The List constructor above initializes a field `head` to point to the initial sentinel of a doubly linked list. The List nodes themselves contain a pointer back to the list as well as `prev` and `next` pointers. As the example shows, we are able to declare all these fields with non-null types and the type checker guarantees that these fields are indeed initialized prior to reading them. Note that when the Node constructor is called with the List argument, the list itself is not properly initialized yet (the `head` field has yet to be assigned). But that is okay, since the Node constructor treats the List parameter as delayed (using the attribute `[Delayed]`), guaranteeing that it won't yet observe any invariants about the list.

2.6.1.2. Non-delayed constructors

Some constructors do more than just initialize fields. They require calling other methods on the object being constructed that require reading of the fields. Such *non-delayed constructors* must be annotated with the attribute `[Microsoft.Contracts.NotDelayed]`. In non-delayed constructors, non-null declared fields must be initialized *prior* to the (implicit or explicit) base constructor call. The base constructor call demarcates the end of non-null declared field initialization. After that call, all non-null declared fields are properly initialized.

In C#, the base constructor call appears always at the beginning of the constructor (often implicitly). This makes it difficult to initialize fields that depend on constructor arguments. In Spec#/Sing#, constructors can call the base constructor explicitly within the body of the method after initializing the non-null declared fields.


```
class C {
   string! name;

   public C(string! n) {
     this.name = n;
     base();  // explicit base call after the non-null field 'name' is initialized
   }
}
```

