Singularity Design Note

6

Exchangeable types

Specifying data formats for messages exchanged over channels

This note describes the low level mechanism and typing support that enables processes to perform 0-copy data exchange. The features are designed to decouple communicating parties and to enable static verification of the hand-off semantics, namely that after the hand-off, the sending party no longer uses the handed-off data.

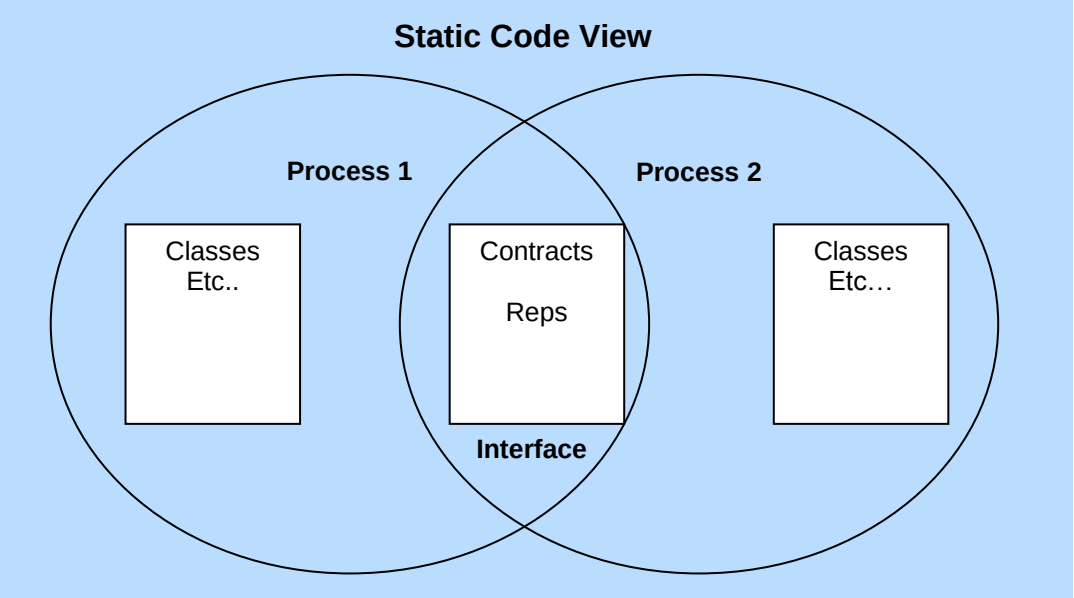
1. Introduction

SDN2 restricts Singularity processes to be closed with respect to executable code. Once a process is started, the entire code base of that process is known and fixed. This has repercussions on the exchange of data between processes. It limits data exchange to known-size data, since data cannot be accompanied by code to interpret it. To illustrate this point, suppose a message containing an object of class B could be sent via a channel. If the sender sends an object *o* of class C (a subclass of B) over the channel, the receiving end could not possibly treat it as a C, since in order to know that there are extra fields and how to access them we would require sending the new methods of C along as well.

Another factor in the design of data exchange is that we desire decoupling the communicating parties as much as possible. Thus, we don't (initially) want to rely on requirements that the client and server agree on some particular version of code for some class C to be exchanged. The picture below shows two process descriptions sharing only an interface consisting of contracts and reps (described in this document). The code bases of these two processes can change independently as long as the interface does not change.

These aspects lead us to the current design which limits the data exchanged over channels to *exchangeable* types. We define exchangeable types and their properties below. The latter sections explore impacts on representation and typing.

Additionally, in order to support 0-copy data exchange we need a form of ownership model of exchangeable blocks of memory. More precisely, we enforce on the sender side, that once a piece (or pieces) of data has been handed off to another process, then the sender has indeed relinquished ownership and does no longer access the sent data. The design enables static enforcement of the ownership transfer.



Decoupling of processes via interfaces.

2. Exchangeable types

A type is *exchangeable (ET)* if it satisfies any of the following conditions:

- It is a scalar
- It is a pointer to a *representation struct* (type R* in ExHeap)
- It is a (by value) representation struct (type R)
- It is a pointer to a variable sized *representation vector* of exchangeable types. (type RepVector<*ET*>)
- It is a (by value) representation vector of exchangeable types of constant size c. (type RepVector<*ET*,c>)

Representation vectors and representation structs (referred to as **rep types**, for short) are new features not present in the current MSIL type system. These are discussed in Section 2.1.

2.1. Representation types

2.1.1. Representation structs

Rep structs are aggregates of exchangeable types, similar to C# structs, but with the following differences:

- A rep struct has no methods, nor constructors
- Fields of a rep struct can only refer to exchangeable types

A block of memory representing a rep struct does not have a GC header. Rep structs can be allocated via a **new** R operation, yielding a 0-initialized untagged block of memory.

2.1.2. Representation vectors

Representation vectors are either

- variable sized of type RepVector<*ET*>, containing elements of exchangeable type *ET*. The actual bit representation of such representation vectors is kept abstract in Singularity but has to be

common among all running processes. This is similar to channel endpoints, whose representation is also controlled by the kernel. A possible representation is a single block of memory with a leading element count.

- fixed size of type RepVector<*ET, Constant*>, containing *Constant* number of elements of exchangeable type *ET*.

Variable sized representation vectors are manipulated as pointers to indirect blocks. Fixed size representation vectors are manipulated by value.

Note the absence of Strings among the exchangeable types. Strings have to be exchanged as variable or fixed length vectors of characters.

2.1.3. Occurrence of rep types

In order to make static verification of ownership transfers feasible, we restrict the occurrence of rep types to

- Locals
- Parameters
- Results
- Fields of rep structs
- Elements of rep vectors
- Message arguments
- Instances of type parameters that are of kind *tracked*. (See Section 5)

Additionally, by-value rep types (inlined rep structs or fixed size rep vectors) can only occur as fields of rep structs or elements of rep vectors
See section 5 for how to refer to rep types from ordinary class objects.

2.1.4. Interior rep pointers

Since rep structs can contain other (inlined) rep structs (as opposed to a pointer to a rep struct), it becomes necessary to obtain pointers to such interior rep structs. Therefore, we support an address-of operation on rep fields inside of reps, yielding a pointer (not a restricted reference).

Since reps cannot be inlined in structs or classes, nor appear inlined on the stack, there is no need to ever have restricted references to a rep.

[TBD: In order to make things simpler for the GC, we might want to restrict interior rep pointers to interior reps starting at the head of the containing rep.]

2.1.5. Rep overlay views

In order to support packet processing, it may be useful to allow having a new view of a prefix of a rep vector or rep struct (or any other part if we allow interior pointers). An overlay is simply a reinterpretation of the bits according to some other rep type R'. E.g., if we have an array of 30 bytes, we could obtain a rep overlay R that interprets the first 8 bytes as a float, and the next 4 bytes as a long. Such an ability would help in writing packet processing code. [See the section on *pointerfree structs* in SDN17 for more on overlays.]

2.2. Properties of exchangeable types

Exchangeable types do not (directly or indirectly) contain any object headers. This property is intentional, since it keeps the exchanged data independent from the particular heap representation of objects chosen by the GCs of the sender and receiver, which in general may use different GCs with differing GC headers. As a result, it is not necessary to adjust any GC headers when blocks of exchangeable data are moved between processes.

On the other hand, the Kernel is responsible for guaranteeing that exchangeable types either have a common representation among all processes in the system, or else are adjusted by the Kernel as part of the transfer to the target process's representation.

2.3. Messages in channel contracts

Messages in channel contracts can only carry arguments of exchangeable type or endpoints. We single out endpoints here to avoid having them sent in nested positions.

2.4. Example

[TBD: give an example…]

3. Rep-Isomorphic classes

Rep structs provide strong isolation between processes, but they are not convenient to program with. On the sender and receiver side we would like to associate code with rep structs independently, so we can take advantage of the usual programming abstractions provided by subtyping, methods, etc. To support such abstraction, we allow processes to declare *rep isomorphic classes*. A rep-isomorphic class uses a particular rep struct as its representation and declares only methods. For example,

```
class MyClassForRepR : rep R, ITracked {


}
```

declares class MyClassForRepR with representation R. Rep-isomorphic classes specify their representation like a base class, but prefixed with the **rep** keyword. They can additionally declare a normal base class, as long as that base-class is abstract and contains no data.

To construct a rep isomorphic object, one starts with a rep pointer r of type R* and calls the automatically generated constructor:

```
MyClassForRepR mr = new MyClassForRepR(r);
```

This operation has the effect of consuming ownership of r, but providing access to r via the new object. References to rep-isomorphic class objects are treated like references to representation structs in that the compiler enforces proper ownership.

In the other direction, one can simply cast a rep-isomorphic reference to a pointer to its representation:

```
R* ptr = (R*)mr;
```

This operation consumes ownership of mr.

In order to implement rep-isomorphic structs, all processes have to agree to allocate rep struct blocks with enough space for a class header (e.g. 12 bytes). The special rep-isomorphic constructor then simply initializes the header bytes of the rep struct block passes as argument.

4. Regions

This section proposes the use of *regions* as a simple initial scheme for managing collections of rep blocks (rep structs and rep vectors). Regions serve both as a well understood operational model, as well as a model for static enforcement of the hand-off semantics. Recall that the hand-off semantics of a send requires that

- Before the send, the sender owns the transmitted reps
- After the send, the sender no longer accesses the transmitted reps

A region serves both as a runtime allocation pool, as well as a type-level name for a group of objects.

4.1. Operational view

Regions are explicit special objects at the IL level with type Region. They support the following operations:

- Allocation of a region
- Allocation of a rep type inside the region
- Manipulating a rep inside a region

- Explicit freeing of a region
- Joining two regions
- Handing off a region via a message send
- Receiving a region via a message receive

The semantics of these operations can be modeled by associating a boolean value **live** with each region indicating whether the region is live in the accessing process, and associating with each rep block *b* the region it lives in **reg**(*b*). The following sections describe the operations in more detail.

4.1.1. Allocation of a region

```
Region r = new Region();
```

The operation creates a fresh region *r*.

post: r.live

4.1.2. Allocation of a rep type inside the region

```
Rep1* rep1 = new Rep1(r);
```

Allocates a fresh rep struct in a given region r.

pre: r.live

post: r.live /\ reg(rep1) == r

Rep vectors are handled similarly:

```
RepVector<Char>* rv = new RepVector<Char>(r, 20);
```

Allocates a fresh rep vector of characters of size 20 in region r.

pre: r.live

post: r.live /\ reg(rv) == r

4.1.3. Manipulating a rep residing in a region

A rep block (vector or struct) residing in a region can be read and written. Both reads and writes have the precondition that the associated region is live.

```
rep1.foo = 55;
```

pre: reg(rep1).live

```
int i = rep1.foo;
```

pre: reg(rep1).live

If the field or element read or written is of rep type, the following additional conditions apply:

```
rep1.foo = rep2;
```

pre: reg(rep1).live /\ reg(rep2) = reg(rep1)

```
Rep2* rep2 = rep1.foo;
```

pre: reg(rep1).live

post: reg(rep2) = reg(rep1)

The operations on rep vectors have analogous conditions. These conditions enforce that everything reachable from a rep in region r also lives in region r. This is a simplifying assumption we are making for the initial version of this proposal. We can later relax this at the cost of a more complicated region model.

4.1.4. Explicit freeing of a region

```
r. Free();
```

Frees region r and all rep blocks contained in it.

pre: r.live

post: not r.live

4.1.5. Joining two regions

```
r1. Join(r2);
```

The operation joins the two regions into a single region. The representative of the joined region is r1, all accesses to region r2 are forwarded to r1.

pre: r1.live /\ r2.live

post: r1.live /\ r2 == r1

Regions may need to be joined due to the restriction that fields of reps and elements of rep vectors point to reps living in the same region as the rep containing the pointer (see also 4.1.3 and 4.2.2). In general, we can think of regions as forming equivalence classes. Initially, each region starts out in its own class and join operations merge classes.

4.1.6. Handing off a region via a message send

```
e. Send (r, rep1, e1, …, en );
```

The operation performs a send on an endpoint e, passing all the data contained in region r. The message object received on the other side is rep1, which must be contained in r (as well as all reps reachable from rep1). Endpoints e1..en sent in the same operations do not interact with regions in any way.

pre: r.live /\ reg(rep1) == r

post: not r.live

4.1.7. Receiving a region via a message receive

```
e. Receive (out r, out rep1, out e1, …, out en );
```

The receiver of a message on endpoint e obtains a region r, and the message object rep1 residing in region r. Dually to the send, the received endpoints do not interact with the received region. The receiver owns the region r and has the same right to it as if the receiver had allocated that region.

post: r.live /\ reg(rep1) == r

4.2. Static enforcement of region model

The goal of the static enforcement of the region semantics is to guarantee at compile-time all the pre conditions of operations involving regions spelled out in the previous section. As a result, we obtain the following benefits:

- We are statically guaranteed that no blocks in dead regions are accessed ever.
- This allows explicit lifetime management of regions (and their content) by the programmer via explicit Free and message send operations.

4.2.1. Simplifying assumptions

To keep the static checking discipline decidable and simple for the programmers we restrict region use as follows[1] :

1. regions cannot be stored in fields of any objects or structs (but see Section 5 on how to relax this).
2. regions are not exchangeable types

Given the semantics of the previous sections, we also know the following condition:

3. a rep block r2 referenced from a rep block r1 (via a field or element of r1) lives in the same region as r1, i.e., reg(r2) == reg(r1).

Because of condition 3 above, we only need to statically track the region of local pointers to rep blocks.
Because of condition 1, we only need to track the liveness of regions in local variables.
Static enforcement (at compile-time) thus tracks the identity and liveness of regions and the mapping between rep blocks and the region they reside in. At any operation involving a region, the associated precondition is statically checked, thereby preventing erroneous operations (like dangling pointer accesses) at compile-time.

4.2.2. Rep writes and region joins

Field assignments defined in the previous sections requires that an object or rep block in region R only points to other rep blocks in region R. The compiler can help the programmer maintain this invariant by inserting region join operations automatically. Suppose we had the following code:

```
Region reg1 = new Region();

Region reg2 = new Region();

Rep1* r1 = new Rep1(reg1);
Rep2* r2 = new Rep2(reg2);

r1.f  = r2;
```

The last assignment requires that reg(r2) is the same region as reg(r1). The compiler can insert the operation

```
r1.Join(r2);
```

in front of the field assignment to maintain the desired invariant, in the case that it would be violated otherwise.

An analogous situation appears if we store a rep pointer into a rep vector at a particular index. In that case the region of the stored pointer is joined with the region of the rep vector.

4.2.3. Message sends

Separating the endpoints from the message, creating the top-level rep block and initializing the message tag, as well as providing a handle on the region for this message is the job of the compiler or runtime system. The programmer would not have to perform these tasks and could instead work with a higher-level message send of the form:

```
e.M(x1,x2,…, xn)
```

[1] We can later relax these decisions at the cost of a more complicated static checking model, requiring more annotations, but providing greater flexibility.

where each xi is either an exchangeable type or an endpoint. The compiler can take care of merging all associated regions among x1..xn, building the single rep struct containing all the non-endpoint arguments and passing the associated region handle to the primitive send operation.

4.2.4. Message receives

Analogous to sends, the compiler can present access to elements in received messages in a uniform way, hiding the special treatment of endpoints and the region handle.

4.2.5. Cross-method support for static analysis

Because of the restrictions made in Section 4.2.1, region objects and rep pointers associated with regions (other than GC) only appear in registers, on the stack, and in parameters and results of methods.

This allows us to elide region information from programs in almost all cases. Within a method, all region associations and liveness information is maintained transparently. At method boundaries, we need some information about what the method does with regions associated with the method parameters of rep type. The method must declare the following:

- What regions are no longer accessible after the method returns: thus far, regions can become inaccessible by callers, if the callee sends the region via a message or frees the region explicitly. A method indicates these situations with a declaration of the form **consumes p**. If p is a rep pointer it indicates that the region associated with p is consumed, otherwise, if p is a region, the region itself is consumed as a result of this method call.
- What regions are being joined into other regions. The declaration **p absorbs q** declares that during the method execution, the region equivalence class of parameter p absorbs the region equivalence class of parameter q.
- In parameters do not usually need to indicate what region they are in, as long as the method treats them without being concerned what region they live in. In order to constrain the region of rep type parameters, one can use the annotation **reg(p) = reg(q)**, where p or q can be rep type parameters or region parameters.
- Which region the out parameters and the result of the method live in: There are several cases:
    1. the result or out parameter p is a region and the callee transfers ownership of the region to the caller. A declaration **grants p** indicates this transfer of ownership.
    2. the result or out parameter is of rep type and the callee transfers ownership of the associated region to the caller. A declaration **grants p** indicates this case. The compiler can provide the new region implicitly as an extra method result.
    3. the result or out parameter lives in the same region as another in or out parameter. In this case a declaration **reg(p) = reg(q)** is used.

  Note that the ownership transfer needs to be declared for each region once only. E.g., if two out parameters p and q of rep type are returned and they both live in the same region, whose ownership is transferred from callee to caller, then the annotations would be **grants p; reg(p) = reg(q)**, rather than **grants p; grants q**. The latter would indicate that p and q live in distinct regions and that ownership of both is transferred.

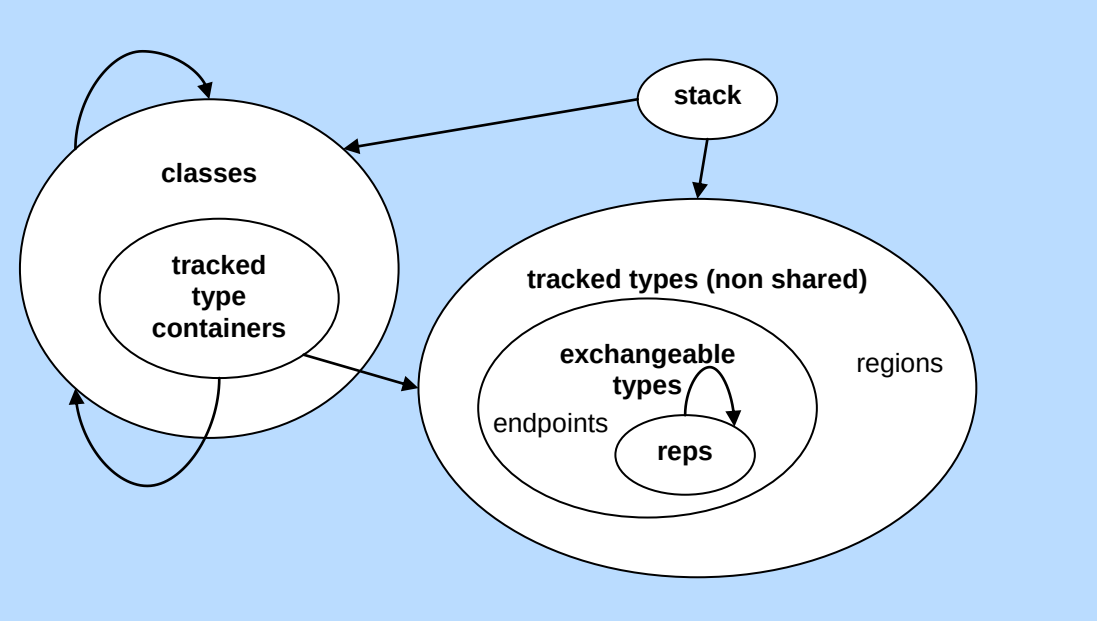
5. Tracked types and containers

Tracked types describe all data that require ownership tracking, namely endpoints, regions, and pointers to reps (in regions). Objects of tracked type are never directly shared among threads.

Ordinary class objects cannot refer to objects of tracked type directly in fields for otherwise ownership tracking would have to extend to ordinary classes. This section discusses how to overcome this serious limitation. The picture below tries to convey the different kinds of data manipulated by a program, as well as the possible references between these kinds of data. At the top-level, we have ordinary class objects (on the left), and tracked objects (on the right). Both kinds can be referenced from the stack. A reference to a group means to any object within that group (even sub groups). Thus, the stack can reference endpoints and reps as well. Ordinary

class objects can hold references to other class objects. (Primitive scalar types are not part of the picture since they can appear everywhere). One sub-group of tracked types is exchangeable types and reps are a sub-group thereof. Reps can refer to other reps.

Ordinary class objects cannot directly refer to tracked types. However, the sub-group *tracked type containers* can refer to tracked types. Tracked type containers are trusted primitive data structures that preserve ownership invariants and also offer simple mutual exclusion among threads. Additionally, tracked type containers also provide the necessary OS operations for endpoint rebinding.



Object classification and possible reference patterns.

Tracked type containers are predefined since they are trusted (hand-verified) to observe the necessary ownership transfer invariants. If the programmer wants to write his own verifiable tracked type containers, we need to have more annotation machinery. We leave this as a future extension. [Ask MAF for details.]

5.1. The simplest tracked type container

The simplest tracked type container is a TREF<T>, containing a single reference to a tracked type. It has the following signature:

```
class TREF<tracked T> {
  public TREF<T>(T x) consumes x;
  public T Acquire() grants result;
  public void Release(T x) consumes x;
};
```

The type parameter T of TREF is marked **tracked** in order to allow this type to be instantiated with tracked types. The constructor takes a T as argument x and indicates that it consumes ownership of x. The Acquire operation returns the tracked type object of the TREF and transfers ownership to the caller (indicated by fresh). Acquire blocks the calling thread if the last operation on the TREF was an Acquire. If a thread tries to Acquire twice in a row, it self-deadlocks. Release returns a tracked type object into the TREF, thus the parameter is again marked as being consumed. Release can be called with the same tracked type object as was obtained by

the preceding Acquire, or with any other object of the correct type T. Release can only be called after an Acquire by the same thread. The correspondence of Acquires and Releases is checked statically and has to occur within a method. A thread could dynamically try to Acquire the same TREF it already has acquired. This situation results in a runtime exception. This possible error is the price to pay for the flexibility of referencing tracked types from arbitrary other objects.

5.1.1. Example

A NameServer needs to map names to service provider endpoints. Since endpoints are tracked types, the hashtable cannot directly map to endpoints. Instead, the table will map from names to TREF< imp<ServiceProvider>>, where imp<ServiceProvider> is an imported endpoint with contract ServiceProvider.

When the server looks up a service, it obtains a TREF<imp<ServiceProvider>> object. It then tries to Acquire the endpoint and interacts with the endpoint (obtaining a new service instance). When the endpoint is back in the initial state of the ServiceProvider contract, the TREF can be Released.

If a second thread of the nameserver tries to serve a request for the same service, while the first thread has Acquired the ServiceProvider, the second thread blocks until the first thread Releases the TREF associated with this service.

5.2. Tracked queue

A slight generalization of the TREF is a queue of tracked types.

```
class TQUEUE<tracked T> {

  public TQUEUE<T>();

  public void Enqueue(T x) consumes x;

  public T Dequeue() grants result;

};
```

Tracked queues store tracked type objects and guarantee that ownership is transferred on an Enqueue from caller to the queue, and on Dequeue from the queue to the caller. The Dequeue operation blocks until an item is available.

6. Copying transfer and non-exchangeable data

If data is to be transferred by copy rather than hand-off, the copy must be made explicitly into a fresh region, and then the copy can be handed-off. Automatic copying of entire regions could be supported by the runtime system if there is such a need.

This section describes how data associated with an ordinary object can be transferred via a channel. Data transfer of ordinary objects is supported via the following steps:

1. allocate a rep block corresponding to the rep to be transferred
2. fill in the rep block with appropriate fields from the object to be transferred
3. send the rep block (hand-off of associated region)
4. the receiver obtains the rep block (and the associated region)
5. the receiver can construct a new object of some class, using the data in the rep block to initialize the necessary fields.

Thus, transferring of object data in principle does not require any particular support, since in order to transfer data for a class C according to representation R, all we need is for C to have the following methods:

- A method returning an R* initialized with the data of C

- A constructor taking an R* as an argument.

These methods essentially act as a serializer and a deserializer to and from representation R.

6.1. Autogenerated serialization

It may be convenient to obtain some simple code to produce reps from a class and vice versa (essentially serialization and deserialization code) automatically. This aspect is beyond the scope of this design note. The Execution Environment team is looking at support for writing code transformations through various tools. Thus serialization would be addressed by a code transformation tool that automatically generates the necessary code for desired classes.

7. Extensions

7.1. Small message optimizations

The implementation for handing off regions via message send may need to be optimized for small messages in such a way that hand-off can happen at granularities less than an entire page.

7.2. Beyond trusted tracked type containers

Verifying code that stores tracked types in fields of classes directly, rather than through predefined trusted tracked type containers is feasible via a discipline of protecting such fields by locks and treating the ownership guarantees as object invariants.

