Singularity Design Note

23

Sing# Implementation details

Interfaces between the Sing# compiler, the runtime system, Bartok, and the verifier.

This note documents the implementation details of features such as rep structs, custom allocators, struct inheritance and the switch-receive interface for the purposes of interfacing with Bartok, the runtime system, and the verifier.

1. Introduction

Two main features that Sing# provides over Spec# are custom allocators for dealing with explicitly managed blocks of memory, as well as a pattern match construct called switch-receive that permits waiting on complex conditions over channels and other selectable events. This note describes in detail how custom allocators are implemented and how the switch-receive implementation works.

2. Rep Structs

Representation structs are a restricted form of struct types that admit unmanaged pointers to them. Since the garbage collector needs to find all managed objects by tracing only through managed pointers, rep structs cannot contain any pointers into the garbage collected heap. In that light, we can view rep structs as a form of unmanaged struct, containing no managed pointers. In Sing#, rep structs are declared with the **rep** prefix:

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

Primitive scalars and enums are considered rep structs as well. Rep structs may contain only other rep structs, or custom heap pointers (see below) to rep structs.
At the MSIL level, rep structs are represented as structs implementing the empty marker interface `System.Compiler.TypeExtensions.IRepStruct`.
In order to facilitate the deallocation of rep struct contents when used in conjunction with custom heaps, the Sing# compiler emits a method called `Dispose` for each rep struct. The Dispose method calls the custom de-allocation routine for any custom heap pointer contained in the struct. Dispose does not free the top-level struct itself.

2.1. Pointer Free Structs

Pointer free structs are a subcategory of rep structs. As the name implies, a pointer free struct contains no pointers whatsoever, only scalars. Pointer free structs are represented at the IL level as rep structs (with the

IRepStruct marker interface). To distinguish them from rep structs, we use a custom attribute [Microsoft.SingSharp.PointerFreeAttribute] on the struct type.

3. Custom Heaps

Custom allocators provide an alternative to GC managed memory. A block of memory allocated in a custom heap is either a single rep struct, or a vector of rep structs. Since custom memory allocators usually need to maintain some meta-data along with the block (e.g., for size information and to link free and used blocks), the Sing# custom allocator design provides a level of abstraction between the logical idea of a pointer to a rep struct in the custom heap, and its actual representation in memory. At the logical level and the IL, such meta-data or header information is not apparent. From the IL perspective, the code manipulates pointers to structs living in a custom heap. The backend compiler Bartok is responsible for bridging the gap between the logical and the physical representation.
For example, in Singularity, we use a single custom heap and corresponding allocator called ExHeap to manage blocks of memory that can be exchanged over channels. The physical representation of a pointer into the ExHeap is a pointer to a header block containing type, size, and ownership information, as well as a pointer to the actual data block. This double indirection is not visible in the IL.

3.1. Types

A custom allocator is simply a class that provides static allocation and deallocation methods, as well as some support for getting at block meta data. At the IL level, we use optional type modifiers to mark pointers into custom heaps and reinterpret some IL instructions if they operate on such pointers.

3.1.1. Custom Allocator Types

A custom allocator is a class deriving from `Microsoft.SingSharp.CustomAllocator`. The latter type is defined in the SingSharp runtime. A custom allocator class CA must correspond to the following template:

```
class CA : Microsoft.SingSharp.CustomAllocator {
  static void * optmod(CA) Allocate(System.Type t);
  static void * optmod(CA[]) AllocateVector(System.Type t, UIntPtr length);
  static void Free(void * modopt(CA) p);

  static UIntPtr GetSize(void * optmod(CA) p);
  static void * IndirectToData(void * optmod(CA) p);
```

The first two methods are allocators for individual blocks or vectors of blocks. The System.Type argument specifies the rep struct type for the purposes of finding the corresponding allocation size. The Free method returns unused allocations (blocks and vectors) to the custom heap for further allocation. The last two methods are used by Bartok to obtain the size of a vector and to indirect past any potential header block. The IndirectToData method permits the memory blocks to be represented with an out-of-band header, where the actual runtime pointer points to a header block, which in turn points to the actual block. This choice of representation abstracted over in the IL. From the IL perspective, the pointer directly points to the data.

3.1.2. Pointer Types

A pointer to a custom heap block is represented at the IL level as a pointer type with an optional modifier. Thus, a pointer to a block representing rep struct R managed by custom allocator CA is encoded using the type:

```
R* optmod(class CA)
```

In order to distinguish at the type level between a single block and a vector of blocks, we use an array type as the modifier. Thus a vector of R's managed by custom allocator CA is encoded using the type:


```
R* optmod(class CA[])
```

Note that the array constructor is on the modifier, not the underlying pointer type. We consider the custom heap vector types compatible with the corresponding non-vector types, i.e., type R* optmod(CA[]) is implicitly convertible to R* optmod(CA).

3.1.3. Conversion to Reference Types

In the IL, managed pointers or refs are used to manipulate pointers to struct in the garbage collected heap (in the form of fields of objects), or pointers to structs on the stack. Such pointers are represented as type

```
R&
```

The IL rules associated with managed pointers are that they cannot be stored into fields, nor returned from methods. This guarantees that they don't escape the scope in which they are generated.
Methods of a struct `R` expect the **this** pointer to have type `R&`, ie., a direct pointer to the memory representing the struct.
We wish to use the same code for manipulating both structs on the stack or in the GC heap, as well as structs in custom heaps. Given that we abstract away the physical representation of custom heap blocks, we have to make a physical distinction between a custom heap pointer and a ref type. Whereas a custom heap pointer may point to a header block or to some word at a particular offset from the actual data, a ref type is a physical pointer to the data. We use an explicit instruction to convert a custom heap pointer to a ref type. This conversion is encoded as a castclass instruction with a target type R&:

```
castclass R&
```

This is not a standard use of castclass and the normal .NET Jitter does not know what to do with such an instruction. For Bartok however, this instruction is well formed, provided the top of the stack contains a pointer of type

```
R* optmod(CA)
```

It is then interpreted as a call to CA::IndirectToData, leaving an R& on the stack.

3.2. Instructions

The following sections show the allocation sequences generated by the Sing# compiler.

3.2.1. Simple allocation sequence

The following Sing# code turns into the IL shown below:

```
R* in CA rp = new[CA] R(1, 2);
```

The code assumes that struct R has a constructor taking two integers. If the default constructor is specified, then no actual constructor call is emitted. We assume allocators zero-initialize all memory.


```
.locals init ([0] valuetype R* modopt(class CA) V_0

ldtoken    R
call       class System.Type System.Type::GetTypeFromHandle(valuetype
                                                            System.RuntimeTypeHandle)
call       void* modopt(class CA) CA::Allocate(class System.Type)
stloc.0
ldloc.0
castclass  valuetype R&
ldc.i4.1
ldc.i4.2
call       instance void R::.ctor(int32, int32)
```

There are two important things to note about the code above. In the IL, pointer to pointer casts are not represented explicitly, in particular, the conversion of the result of the allocation from void* optmod(CA) to R* optmod(CA) is implicit in the stloc.0 instruction, given that the local variable 0 has type R* optmod(CA). Bartok depends on such pointer casts to be represented as stores to locals with the appropriate target type. The second thing to note is how the custom heap pointer is converted to type R& in order to make it compatible with the receiver type of the constructor of R.

Verifiability

The allocation sequence verification requires that the ldtoken type agrees with the pointer target type of the local into which the allocation result is stored.

3.2.2. Vector allocation

Vector allocation is analogous to single blocks, except that no constructors are invoked.

```
.locals init ([0] valuetype R* modopt(class CA[]) rv)
ldtoken    R
call       class System.Type System.Type::GetTypeFromHandle(valuetype System.RuntimeTypeHandle)
ldc.i4.s   20
call       void* modopt(class CA[]) CA::AllocateVector(class System.Type, native unsigned int)
stloc.0
```

Verifiability

The allocation sequence verification requires that the ldtoken type agrees with the pointer target type of the local into which the allocation result is stored.

3.2.3. De-allocation

Single block and vector de-allocation are generated via the following code:

```
delete p;
```

Provided that p has type R* opt(CA), the following IL is generated.

```
ldloc.2
castclass  valuetype R&
call       instance void R::Dispose()
ldloc.2
call       void CA::Free(void* modopt(CA))
```

where local 2 has type


```
.locals init ([2] valuetype R* modopt(class CA) rp)
```

The call to Dispose recursively frees any non-null custom heap pointer contained within the struct R itself. For vectors, the de-allocation sequence does not contain the Dispose call.
TODO: we must insert a loop to call Dispose on each element.

Verifiability

The verification requires that the argument to Free is owned by the current stack frame at the point of call to Free.

3.2.4. Field access

Accessing fields of structs via custom heap pointers is performed using the normal ldfld and stfld instructions of the IL. Bartok does not require the top of the stack to be an R&. If the top of the stack is of type R* optmod(CA), then the call to CA::IndirectToData is inserted by Bartok.

Verifiability

The custom heap pointer must be live at the point of the field read/write.

3.2.4.1. Ldflda

This instruction computes the address of a field. Bartok performs the indirection to data using a call to CA::IndirectToData and then leaves a physical pointer of type S& on the stack, provided the field has type S.

Verifiability

The custom heap pointer must be live at the point of the field read/write. Furthermore, the lifetime of the heap pointer must extend beyond the lifetime of the result of type S& (since it is an interior pointer).

3.2.5. Initobj, stobj, ldobj

These instructions work on values of custom heap pointer type as they do on ref and normal pointer types. Bartok is responsible for inserting calls to CA::IndirectToData.

Verifiability

The custom heap pointer must be live at the point of the operation.

3.2.6. Vector operations

We reuse the vector instructions ldlen, stelem, ldelem, ldelema that are defined to operate on single dimensional arrays by the .NET platform. For Bartok, these instructions are well formed, if the top of the stack contains a custom heap pointer of type R* optmod(CA[]).

3.2.6.1. ldlen

Bartok inserts a call to CA::GetSize and leaves the size native int on the stack.

Verifiability

The custom heap pointer must be live at the point of the operation.

3.2.6.2. ldelem, stelem

Bartok is responsible for generating code that uses the size of the struct R, and the index to compute the appropriate physical offset, performs a runtime bounds check (throwing IndexOutOfBounds), and performs the actual memory load or store (after using CA::IndirectToData).

Verifiability

The custom heap pointer must be live at the point of the operation.

3.2.6.3. ldelema

This instruction performs the same offset computation and bounds checking as the ldelem and stelem instructions, but leaves the physical address of the addressed element on top of the evaluation stack. The result has type R&.

Verifiability

The custom heap pointer must be live at the point of the ldelema. Furthermore, the lifetime of the heap pointer must extend beyond the lifetime of the result of type R& (since it is an interior pointer).

3.2.6.4. box

The normal box instruction creates an object in the GC heap and copies the contents of a struct value into that object. We add a new interpretation of the box instruction when the top of the stack is a custom heap pointer. In that case, Bartok generates code that creates a box containing the pointer to the custom heap object. This instruction is useful to view a custom heap block as a common abstraction.

Verifiability

The instruction is verifiable, if custom heap pointer on the stack is live and the target type of the box is an interface type I deriving (directly or indirectly) from marker interface Microsoft.SingSharp.IBorrowed. The idea behind IBorrowed is to put a store restriction similar to ref type on such objects that prevent them from being returned from methods and stored into fields. Naturally, there's also the restriction that prevents upcasts from IBorrowed to System.Object.
Furthermore, the lifetime of the box must be contained within the lifetime of the original custom heap object (since it is essentially an alias).

3.2.6.5. castclass R&

As discussed earlier, we reinterpret the castclass instruction if the target type is of type R& to mean a conversion from the logical custom heap pointer to the physical representation. Thus Bartok must insert a call to the custom allocator IndirectToData method. The result on the stack is of type R&.

Verifiability

The custom heap object on top of the stack must be live and the lifetime of the result of type R& must be contained within the lifetime of the original custom heap object.

3.2.7. Struct Inheritance

4. Switch-Receive

Table of Contents

