Charting new territory: bytecode verification by type-checking, overflow checking, unsigned integrals, and a nullables primer

© Miguel Garcia, LAMP, EPFL
http://lamp.epfl.ch/~magarcia

September 1st, 2010

Contents

1 Inserting safe `castclass` to keep `peverify` happy

1.1 Example

Can we in `forMSIL`-mode add those redundant casts? They would unclutter the `peverify` output (code runs ok without them).

Many examples (also in other compilers[1]). One of them:

```
[IL]: scala.util.parsing.input.OffsetPosition::gd1$1
      [offset 0x00000009][found value 'java.lang.CharSequence']
      Unexpected type on the stack.(Error: 0x80131854)

.method public hidebysig virtual instance class scala.collection.immutable.Map
        updated(object key,
                object 'value') cil managed
{
  // Code size       164 (0xa4)
  .maxstack 8
  .locals init (object V_0,
           object V_1)
  . . .
  IL_009e: newobj     instance void scala.collection.immutable.Map$/Map3::.ctor(object,
                                                                                object,
                                                                                object,
                                                                                object,
                                                                                object,
                                                                                object)
  IL_00a3: ret /*- for those in the know, class scala.collection.immutable.Map$/Map3
                   does implement the return type of this method,
                   but peverify seems not to be one of them ... */
} // end of method Map2::updated
```

1.2 How peverify merges types

The relevant spec fragment is §*III.1.8.1.3 Merging stack states*:

> *the overall merge shall be computed by merging the states slot-by-slot as follows. Let `T` be the type from the slot on the newly computed state and `S` be the type from the corresponding slot on the previously stored state. The merged type, `U`, shall be computed as follows (recall that `S := T` is the compatibility function defined in §1.8.1.2.2):*
>
> *1. if `S := T` then `U=S`*
> *2. Otherwise, if `T := S` then `U=T`*
> *3. Otherwise, if `S` and `T` are both object types, then let `V` be the closest common supertype of `S` and `T` then `U=V`.*
> *4. Otherwise, the merge shall fail.*
>
> *Merging a controlled-mutability managed pointer with an ordinary (that is, non-controlled-mutability) managed pointer to the same type results in a controlled-mutability managed pointer to that type.*
>
> *Implementation Specific (Microsoft): The V1.0 release of the Microsoft CLI will merge interfaces by arbitrarily choosing the first common interface between the two verification types being merged.*

The implementation of the the above in CCI's `TypeHelper` is:

```
/// <summary>
/// Returns the merged type of two types as per the verification algorithm in CLR.
```

[1] http://connect.microsoft.com/VisualStudio/feedback/details/96401/peverify-false-unexpected-type-on-the-stack-on-c-ternary-operator-when-mixing-array-and-ilist1-types

Listing 1: Sec. 1.2

```
def nearestSuperclass(type1: Type, type2: Type): Type = {
  var depth1 : Int = 0
  var t1 : Type = type1
  var typeIter = t1
  while (typeIter != null) {
    typeIter = if (typeIter.parents.isEmpty) null else typeIter.parents.head
    depth1 = depth1 + 1
  }
  var depth2 : Int = 0
  var t2 : Type = type2
  typeIter = t2
  while (typeIter != null) {
    typeIter = if (typeIter.parents.isEmpty) null else typeIter.parents.head
    depth2 = depth2 + 1
  }
  while (depth1 > depth2) {
    t1 = t1.parents.head
    depth1 = depth1 - 1
  }
  while (depth2 > depth1) {
    t2 = t2.parents.head
    depth2 = depth2 - 1
  }
  while (depth1 > 0)
  {
    if (t1 == t2)
      return t1
    t1 = t1.parents.head
    t2 = t2.parents.head
    depth1 = depth1 - 1
  }
  return null
}
```

```
/// </summary>
//^ [Pure]
public static ITypeDefinition MergedType(ITypeDefinition type1, ITypeDefinition type2) {
  if (TypeHelper.TypesAreAssignmentCompatible(type1, type2))
    return type2;
  if (TypeHelper.TypesAreAssignmentCompatible(type2, type1))
    return type1;
  ITypeDefinition/*?*/ lcbc = TypeHelper.MostDerivedCommonBaseClass(type1, type2);
  if (lcbc != null) {
    return lcbc;
  }
  return Dummy.Type;
}
```

In turn, the Scala formulation of `MostDerivedCommonBaseClass` can be found in Listing 1.

Also in CCI, the metadata model to code model decompiler has visitor called `Unstacker` that computes type-stacks. See project `http://cciast.codeplex.com/`.

1.3 A first attempt

There's a lot of knowledge about lubs that `GenICode` does not record in the ICode instruction stream (tentative idea: how about having a pseudo-instruction `VERIF_CAST` in addition to `CHECK_CAST` to track that information?). Coming back to the `if-then-else` example, and patching `genLoadIf` a bit:

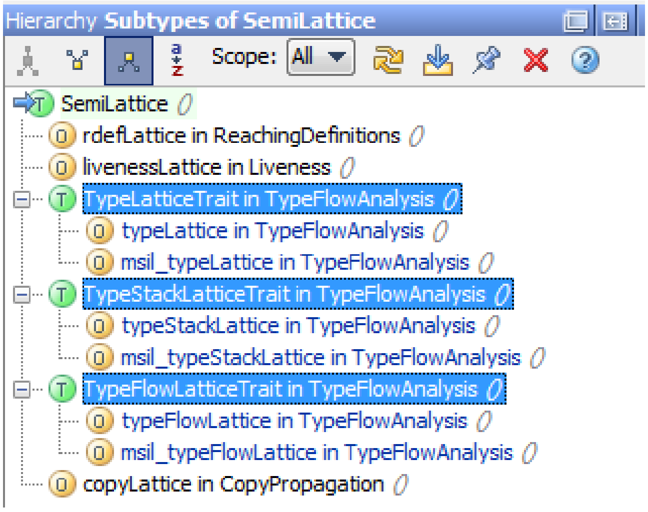
```
private def genLoadIf(tree: If, ctx: Context, expectedType: TypeKind): (Context, TypeKind) = {
  val If(cond, thenp, elsep) = tree

  var thenCtx = ctx.newBlock
  var elseCtx = ctx.newBlock
  val contCtx = ctx.newBlock

  genCond(cond, ctx, thenCtx, elseCtx)

  val ifKind = toTypeKind(tree.tpe)
  val thenKind = toTypeKind(thenp.tpe)
  val elseKind = if (elsep == EmptyTree) UNIT else toTypeKind(elsep.tpe)

  /*- CLR LUB START */
  val lubAsPerPEVerify = icodes.msil_lubPEVerify(thenKind, elseKind)
  if (!(lubAsPerPEVerify <:< ifKind)) {
    // ifKind more specific than lubAsPerPEVerify
    contCtx.bb.emit(VERIF_CAST(ifKind))
  }
  /*- CLR LUB END */
```

1.4 Custom type-merge functions while reusing the type-flow framework

A straightforward way to extend the forward-dataflow framework (in package `scala.tools.nsc.backend.icode.analysis`) consists in directly subclassing `SemiLattice` (usually by an `object`) and `DataFlowAnalysis` (where the `val lattice` in the subclass is overriden with the `object` just mentioned). Current extensions following this pattern:

- `livenessLattice` and `LivenessAnalysis`,
- `rdefLattice` and `ReachingDefinitionsAnalysis`,
- `copyLattice` and `CopyAnalysis`,
- the type lattice, type stack lattice, and type flow lattice (objects) and the `MethodTFA extends DataFlowAnalysis` class.

We want to explore how to reuse most of `MethodTFA` while using a custom `lub` function. After moving to traits the functionality we want to reuse:



a different `lub` function can be used as follows:

```
class MsilMethodTFA extends MethodTFA {
  override val lattice = msil_typeFlowLattice
   /* override def interpret(in: analysis.typeFlowLattice.Elem,
                             i: Instruction): analysis.typeFlowLattice.Elem =
         super.interpret(in, i) */
}
```

Overriding other members (e.g., `interpret`) is also possible but in the example not necessary. The only change performed,

```
object msil_typeFlowLattice extends {
  val myTypeStackLattice = msil_typeStackLattice
  val myTypeLattice    = msil_typeLattice
} with TypeFlowLatticeTrait
```

in fact boils down to this:

```
object msil_typeLattice extends TypeLatticeTrait {

  override def lub2(exceptional: Boolean)(a: Elem, b: Elem) =
    if (a eq bottom) b
    else if (b eq bottom) a
    else icodes.msil_lubPEVerify(a, b)

}
```

1.5 Similar (identical?) problem: computing stack-map frames for `jvm6`

Current effort on the backend for jvm-6 : `http://github.com/dragos/scala/tree/backend-jvm6` Can be tested by building the compiler with '`-target:jvm6`' and then running the produced compiler with java `-XX:-FailOverToOldVerifier` (to disable the fallback to the old verifier). A few highlights:

- http://github.com/dragos/scala/tree/backend-jvm6/src/compiler/scala/tools/nsc/backend/icode/analysis/
- file `VerificationTypes.scala` in that folder

Kinds of problems to solve, quoting from an email:

```
abstract class Base { def foo = 10 }
trait T extends Base { def bar = 20 }

class D1 extends T {}
class D2 extends T {}

object Test extends Application {
  val x = null
  val y = if (x ne null) new D1 else new D2
  y.foo /*- call 1 */
  (if (x ne null) new D1 else new D2).foo /*- call 2 */
}
```

> *The first call goes through, because* `erasure` *adds a cast to* '`Base`' *before the call. The second call fails. The problem is that a trait that extends a class, is translated in the bytecode to extend* `Object`*. That is because traits are interfaces, and it is impossible to extend a non-interface. The lub of* `D1` *and* `D2` *is then* `Object` *(in the bytecode), although in Scala it would be trait* `T`*, and the call to* `foo` *is well-typed.*

Sidenotes:

- Updated class format specification (JSR 202)
- There are a few "Type Map Inference Tools" to bring classfiles from older classfiles versions to 50.0, for example:
  - ASM 3.1: `org.objectweb.asm.tree.analysis` provides a static bytecode analysis framework on top of the tree package. It can be used in addition to the tree package to implement really complex class transformations that need to know the state of the stack map frames for each instruction. Details at http://asm.ow2.org/doc/developer-guide.html#controlflow
  - Computing stack maps with interfaces, Frédéric Besson, Thomas Jensen, and Tiphaine Turpin, http://www.irisa.fr/celtique/fbesson/Computing_Stack_Maps_With_Interfaces.pdf
  - ProGuard preverification
- Given a `.class` file, creates and injects `StackMapTable` attributes in it. http://code.google.com/p/yasmit/. Written in Haskell.

1.6 Another prototype

A compiler plugin running after `GenICode` invokes a `MethodTFA`-like analysis that sports a type-merge function as in `peverify`. This by itself does not insert the required casts, but helps in visualizing disagreements between what `peverify` guesses to be on the evaluation stack, what `MethodTFA` guesses, and what the signatures expect (say, in the method signature for a `CALL_METHOD`).

```
val icode_tfa = new analysis.MsilMethodTFA
val peverif_tfa = new analysis.PEVerifMethodTFA
```

Some examples (thanks Iulian!):

```
abstract class Base { def foo = 10 }
trait T extends Base { def bar = 20 }

case class D1 extends T {}
case class D2 extends T {}

object Test extends Application {
  val c: Boolean = true
  def moo(x: Base) {}
  moo(if (c) new D1 else new D2)
}
```

The prototype reports the differences in type-stack guesses between ICode and PEVerify. In particular for the invocation on "`moo`" those guesses are:

```
  block: 4
     pre icode  typestack : [REFERENCE(scala.Product),REFERENCE(Test),REFERENCE(System.Object)]
     pre peverif typestack : [REFERENCE(Base),    REFERENCE(Test),REFERENCE(System.Object)]
  0| CALL_METHOD TestTest.moo (dynamic)
     post type stacks match : [REFERENCE(System.Object)]
  1| RETURN (UNIT)
     post typestacks match : [REFERENCE(System.Object)]
```

As of now, the `peverify`-like guess is not updated to reflect future usages of the stack slot. For example, in case the invoked method has a formal parameter with an interface type:

```
abstract class Base { def foo = 10 }
trait T extends Base { def bar = 20 }

case class D1 extends T {}
case class D2 extends T {}

object Test extends Application {
  val c: Boolean = true
  def moo(x: Base) {}
  def moo2(x: Product) {}
  moo(if (c) new D1 else new D2)
  moo2(if (c) new D1 else new D2)
}
```

Running the compiler plugin will report still the same (although now `moo2` expects a `Product` argument):

```
  block: 7
     pre icode  type stack : [REFERENCE(scala.Product),REFERENCE(Test),REFERENCE(System.Object)]
     pre peverif type stack : [REFERENCE(Base),    REFERENCE(Test),REFERENCE(System.Object)]
  0| CALL_METHOD TestTest.moo2 (dynamic)
     post type stacks match : [REFERENCE(System.Object)]
  1| RETURN (UNIT)
     post type stacks match : [REFERENCE(System.Object)]
```

because the lub algorithm in use takes into account the base class hierarchy only, giving "`Base`" because the IClasses to be instantiated in the if-then-else branches are:

```
class D1 extends Base, T, ScalaObject, Product

class D2 extends Base, T, ScalaObject, Product
```

1.7 TODO

In order to insert casts, the prototype should additionally track:

- which `(bb, idx)` pushes each slot (see `ReachingDefinitions` in package `scala.tools.nsc.backend.icode.analysis`. The reaching definitions analysis was covered in a previous write-up[2]).
- in case the type required (at the point of use) does not cover that guessed as per `peverify`'s algorithm, the information about provenance can be used to insert casts right after the load instruction.
- For an input program well-typed, the inserted casts will not fail at runtime. I haven't thought in detail about forward/backward jumps but the above still sounds plausible to me.

In the `moo2` example, the cast into `Product` would be inserted right after loading what will be the argument to `moo2`'s invocation (one cast per branch of the `if-then-else`).

2 Overflow checking and the CIL for that

Much like Scala's `synchronized`, there's in the C# `checked` operator (a keyword actually) to make the directly enclosed expression or block throw a `System.OverflowException` in case of over/underflow for integral arithmetic operations. The notes below summarize both syntax and CIL aspects of this capability of the CLR, which has no counterpart in the JVM. The C# spec'ed behavior mentions that invocations are not affected by the `checked`/`unchecked` status at the callsite, so I guess the same should apply in case of inlining. Easy solution: never let the compiler inline, just the VM.

Quoting from the C# 3.0 lang spec [2, §7.5.12]

> *The checked and unchecked operators are used to control the overflow checking context for integral-type arithmetic operations and conversions.*
>
> *checked-expression:*
> *checked (expression)*
>
> *unchecked-expression:*
> *unchecked (expression)*

[2] http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/threeaddress.pdf

The `checked` *operator evaluates the contained expression in a checked context, and the* `unchecked` *operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression (§7.5.3), except that the contained expression is evaluated in the given overflow checking context. The overflow checking context can also be controlled through the checked and unchecked statements (§8.11).*

Details about the built-in operations affected by overflow checking are given in [2, §7.5.12]. The relevant CIL instructions are:

- instructions of the `add`, `sub`, and `mul` family, which can be qualified to perform `.ovf` checking (these instructions also exhibit `.un` variants to operate on unsigned integrals).
- the `conv.<toType>`, `conv.ovf.<toType>`, and `conv.ovf.<toType>.un` instructions.

2.1 The `div` special case

The spec shows an irregularity of `div` with respect to (`add, sub, mul`) and (`add.ovf, sub.ovf, mul.ovf`):

Stack Transition*:* ..., *value1, value2* → ..., *result*

Exceptions*:*

Integral operations throw `System.ArithmeticException` *if the result cannot be represented in the result type. (This can happen if value1 is the smallest representable integer value, and value2 is -1.)*

Integral operations throw `DivideByZeroException` *if value2 is zero.*

Implementation Specific (Microsoft): On the x86 an `System.OverflowException` ***is thrown when computing (***`minint div -1`***).***

Floating-point operations never throw an exception (they produce NaNs or infinities instead, see Partition I).

Stated another way (quoting from the IKVM blog):

Why didn't they define a `div.ovf` *(like there are* `add.ovf, sub.ovf, mul.ovf`*) in addition to* `div` *(and then make* `div` *behave consistently with* `add, sub, mul`*)?*

3 Unsigned integrals and Scala.NET

Coming back to one of the issues we saw in the `Bootstrapping4.pdf` write-up:

```
public static hidebysig System.Int64 System.Math::DivRem(System.Int64,
                                                         System.Int64,
                                                         System.Int64& /*- problem for TypeParser */
                                                         )
```

3.1 Background on CLR unsigned integrals

Because it does not mess with memory safety, signed and unsigned integrals of the same size are assignment-compatible in the CLR. No conversion is required, and no runtime check is performed for values-out-of-range. We can happily place a `scala.Byte` where a `System.Byte` was expected (i.e., pass a signed argument to an unsigned parameter).

For the same token, CIL arith instructions "interpret" the operands on the stack as being signed or unsigned depending on the instruction variant used, irrespective of the "sign" with which those operands were pushed, or whether one but not the other is signed for that matter. Additional highlights can be found in *PartitionIII.1.1.1*.

3.2 `int[]` is `uint[]` in C# (syntax lumps together casts and coercions)

Quoting from StackOverflow:

```
// C# code
public void Test()
{
    object intArray = new int[] { -100, -200 };
    /*-
        'intArray is uint[]' below
        wouldn't typecheck
        without 'object' above
    */
    if (intArray is uint[]) //why does this return true?
    {
        uint[] uintArray = (uint[])intArray; //why no class cast exception?
        for (int x = 0; x < uintArray.Length; x++) { Console.Out.WriteLine(uintArray[x]); }
    }
}
```

Quoting from the CLR spec:

> *8.3.2 Coercion*
>
> *Coercion takes a value of a particular type and a desired type and attempts to create a value of the desired type that has equivalent meaning to the original value. Coercion can result in representation changes as well as type changes; hence coercion does not necessarily preserve the identity of two objects. There are two kinds of coercion: widening, which never loses information, and narrowing, in which information might be lost. An example of a widening coercion would be coercing a value that is a 32-bit signed integer to a value that is a 64-bit signed integer. An example of a narrowing coercion is the reverse: coercing a 64-bit signed integer to a 32-bit signed integer. Programming languages often implement widening coercions as implicit conversions, whereas narrowing coercions usually require an explicit conversion. Some widening coercion is built directly into the VES operations on the built-in types (see §12.1). All other coercion shall be explicitly requested. For the built-in types, the CTS provides operations to perform widening coercions with no runtime checks and narrowing coercions with runtime checks.*

8.3.3 Casting

Since a value can be of more than one type, a use of the value needs to clearly identify which of its types is being used ... Unlike coercion, a cast never changes the actual type of an object nor does it change the representation. Casting preserves the identity of objects.

The following post by Eric Lippert[3] refers to the way C# wallpapers the above:

- *First source of confusion: in C# we have conflated two completely different operations as "cast" operations. The two operations that we have conflated are what the CLR calls casts and coercions.*

  We conflate these two things in C#, using the same operator syntax and terminology for both casts and coercions. So now it should be clear that there is no cast from `int` *to* `float` *in the CLR. That's a coercion, not a cast. Second source of confusion: inconsistency in the CLR spec. The CLR spec says in section 8.7*

  Signed and unsigned integral primitive types can be assigned to each other; e.g., `int8 := uint8` *is valid. For this purpose,* `bool` *shall be considered compatible with* `uint8` *and vice versa, which makes* `bool := uint8` *valid, and vice versa. This is also true for arrays of signed and unsigned integral primitive types of the same size; e.g.,* `int32[] := uint32[]` *is valid.*

  And in section 4.3:

  If the class of the object on the top of the stack does not implement class (if class is an interface), and is not a derived class of class (if class is a regular class), then an `InvalidCastException` *is thrown.*

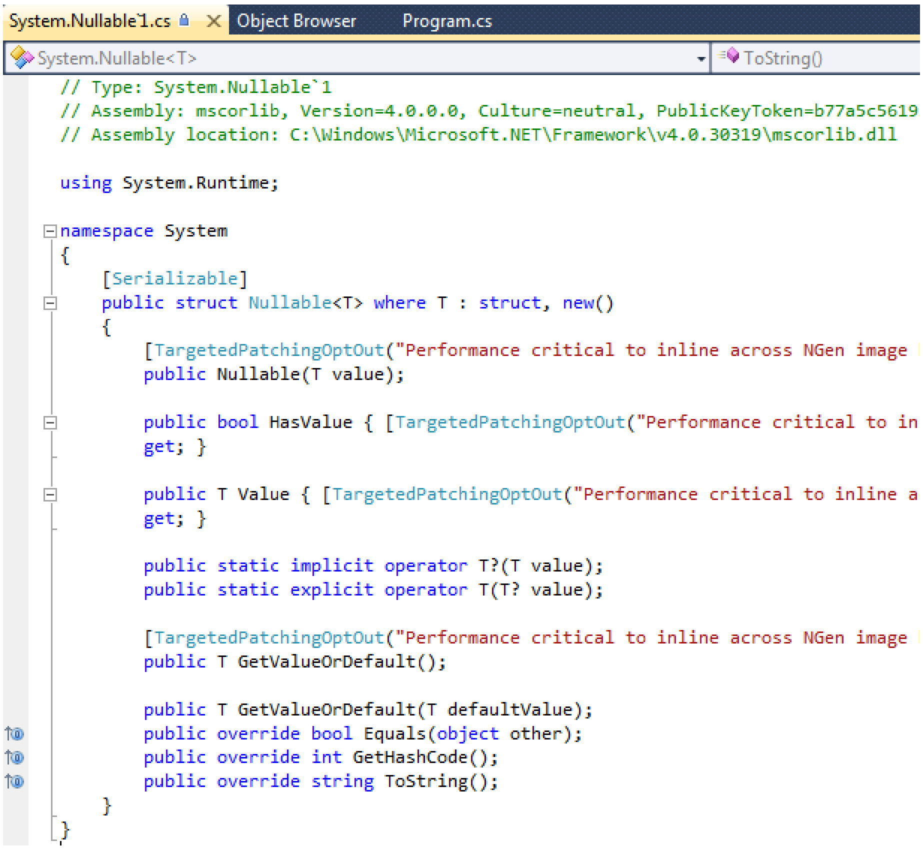
- *Second source of confusion: If* `Foo` *can be cast to* `Bar`*, then* `Foo[]` *can be cast to* `Bar[]`*.*

Where does the spec for `castclass` ***say that*** `int32[]` ***can be cast to*** `uint32[]`***? It doesn't. It should!*** `int32` ***and*** `uint32` ***are assignment compatible, so they can be cast from one to the other without changing bits. But they do not implement or derive from each other.***

Casting between assignment-compatible types should be legal. Really what this should say is something like "If `Foo` *can be cast to* `Bar` *or* `Foo` *is assignment compatible with* `Bar` *then* `Foo[]` *can be cast to* `Bar[]`*". Fortunately, the CLR guys did NOT extend this goofy kind of type variance to covariant and contravariant interfaces, which as you know we are probably adding in a future version of C#. That is, if we make* `IEnumerable<T>` *covariant in T, it will NOT be possible to do a clever series of casts to trick the CLR into assigning an* `IEnumerable<int>` *to an* `IEnumerable<uint>`*, even though it is possible to make* `int[]` *go to* `uint[]`*. However, I think it is possible (I haven't checked this yet) to leverage the fact that* `int[]` *goes to* `uint[]` *to similarly force* `IEnumerable<int[]>` *to go to* `IEnumerable<uint[]>`*.*

---

[3] http://groups.google.com/group/microsoft.public.dotnet.languages.csharp/browse_thread/thread/2d21bf036a23918e



```
System.Nullable`1.cs   Object Browser   Program.cs
System.Nullable<T>   ToString()
// Type: System.Nullable`1
// Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c5619
// Assembly location: C:\Windows\Microsoft.NET\Framework\v4.0.30319\mscorlib.dll

using System.Runtime;

namespace System
{
    [Serializable]
    public struct Nullable<T> where T : struct, new()
    {
        [TargetedPatchingOptOut("Performance critical to inline across NGen image
        public Nullable(T value);

        public bool HasValue { [TargetedPatchingOptOut("Performance critical to in
        get; }

        public T Value { [TargetedPatchingOptOut("Performance critical to inline a
        get; }

        public static implicit operator T?(T value);
        public static explicit operator T(T? value);

        [TargetedPatchingOptOut("Performance critical to inline across NGen image
        public T GetValueOrDefault();

        public T GetValueOrDefault(T defaultValue);
        public override bool Equals(object other);
        public override int GetHashCode();
        public override string ToString();
    }
}
```

Figure 1: `[mscorlib]System.Nullable‘1<T>`

This situation of the CLR being more generous about what identity-preserving casts are legal may end up considerably complicating my life in other ways involving covariance and contravariance as we attempt to detect ambiguous conversions at compile time, but that's another story and we are still researching it.

4 What makes `Nullable<T>`s (slightly) different from other valuetypes

The snippet below and its ILAsm counterpart (Listing 2) show a few translation idioms, discussed in the followin subsections.

```
static void NullablesSample()
{
    int? ni1 = null;
    int? ni2 = 10;
    int nres = (int)(ni1 + ni2);
}
```

Nullable values are created (Sec. 4.1) as for other valuetypes, while their use in C# ("propagate nulls") involves syntax sugar to reproduce monadic style (Sec. 4.2). Talking about syntax sugar, the C# type-ref "int?" can be expressed in Scala for example as ?[Int], after aliasing type ?[T <: System.ValueType] = System.Nullable[T].

4.1 Initialization and assignment

Creating values of nullables is no different from other valuetypes. One may:

- start with a managed pointer and invoke either innitobj or call the single-arg constructor; or
- start with a raw value and newobj to obtain a nullable on the stack. The same effect can be achieved with the static op_Implicit of System.Nullable‘1<T>.

An assignment to a nullable (Figure 1)may have as as RHS any of: null, a compatible raw value, or a compatible nullable. Examples follow for the first two cases (the third involves copying object references, in the example ldloc.s ni1; stloc.s ni2 would achieve that).

- the shortest instruction sequence that assigns null involves initializing in place to the "default value": load a managed pointer for the LHS, followed by initobj with [mscorlib]System.Nullable‘1<V> as *typeTok* (where V is the non-nullable valuetype in question)
- there's no no-args constructor for nullables, and the single-arg constructor (invoked directly below) expects a raw value as argument (i.e., null can't be loaded). Like this:

```
IL_0009: ldloca.s  ni2
IL_000b: ldc.i4.s  10
IL_000d: call      instance void valuetype [mscorlib]System.Nullable‘1<int32>::.ctor(!0)
```

- there's a static method receiving a raw value that leaves a nullable on the stack, op_Implicit, used as follows:

```
IL_0009: ldc.i4.s  10
IL_000b: call      valuetype [mscorlib]System.Nullable‘1<!0> valuetype [mscorlib]System.Nullable‘1<int32
```

In turn, op_Implicit is defined as follows:

```
.method public hidebysig specialname static
       valuetype System.Nullable‘1<!T> op_Implicit(!T 'value') cil managed
{
  // Code size     7 (0x7)
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: newobj    instance void valuetype System.Nullable‘1<!T>::.ctor(!0)
  IL_0006: ret
} // end of method Nullable‘1::op_Implicit
```

BTW, attempting to directly assign null to a nullable (ldnull, stloc for example) results in an "Unexpected type on the stack" error ("[found Nullobjref

'NullReference'][expected value 'System.Nullable`1[System.Int32]']"). The very same error is raised by `peverify` for the instructions:

```
IL_0001: ldnull
IL_0002: box    valuetype [mscorlib]System.Nullable`1<int32> /*- fails to pass peverify */
```

```
IL_0001: ldc.i4.s  10
IL_0002: box    valuetype [mscorlib]System.Nullable`1<int32>
/*- fails too: [found Int32][expected value 'System.Nullable`1[System.Int32]'] */
```

4.2 No boxing, no unboxing, but is it monadic style? Really?

The statement `int nres = (int)(ni1 + ni2);` in the example gets translated into the pseudocode below (full details in Listing 2). Given the null-value in `ni1`, an `InvalidOperationException` will be thrown.

```
IL_0014: C0 := ni1
IL_0016: C1 := ni2
IL_0027: if (C0.HasValue && C1.HasValue) {
           push ( C0.GetValueOrDefault() + C1.GetValueOrDefault() )
         } else {
           C2 := null;
           push C2;
         }
IL_0049: pop   C2
IL_004b: res := C2.get_Value()
IL_0053: ret
```

Samples of monadic style in C# (the Maybe monad, nullables in particular):

- Eric Lippert, Monads in plain English[4].
- The example below is reproduced from Wes Dyer's article on the subject[5].

```
var r = from x in 5.ToMaybe()
        from y in Maybe<int>.Nothing
        select x + y;

// Console.WriteLine(r.HasValue ? r.Value.ToString() : "Nothing");
// would display "Nothing"
```

And now, all over in Scala:

- Burak Emir, Monads in Scala: http://lamp.epfl.ch/~emir/bqbase/2005/01/20/monad.html
- arithmetic with `Options`, http://quoiquilensoit.blogspot.com/2009/11/using-arithmetic-expression-with-option.html

[4] http://stackoverflow.com/questions/2704652/monad-in-plain-english-for-the-oop-programmer-with-no-fp-backgr 2704795#2704795

[5] http://blogs.msdn.com/b/wesdyer/archive/2008/01/11/the-marvels-of-monads.aspx

4.3 How compilers special-cases `Nullable` (a *bad thing*)

4.3.1 F#

There's no syntax sugar for nullable types in F#[6], because they don't correspond cleanly to the `Option` abstraction. Quoting from `http://connect.microsoft.com/VisualStudio/feedback/details/470052/f-nullable-t-vs-option-t`, (January 2009, by Luke Hoban, F# Program Manager):

> *`Option` is designed to act as a type-explicit representation of a no-value-present condition. It is used to augment any existing type with a single additional value. `Nullable`, on the other hand, was designed to provide optional nullability for .NET struct types, to ensure that null was a consistently available value across both reference types and (nullable) struct types – in particular for O/R mapping scenarios.*
>
> *The `Nullable` type pretty fundamentally does not accomplish the design goal of `Option` though – its struct constraint means that it cannot add an additional value to anything but a struct type, and so cannot be used to represent the no-value-present condition in any generic functions.*
>
> *For example, the F# function `List.tryFind` is a common operation for searching a `List` using a given predicate. It can return either an element of the list, or nothing. This requires a type satisfying the `Option` design criteria to encode the return value – a type which has all the values of the List element type, plus one to represent "not found".*
>
> *There is room to provide further `Nullable` support in the F# language, and much of this can be done today through user defiend operators and conversions. We expect to look at doing more of this in a future version of F#.*

Regarding arithmetic involving nullables[7]:

> *F# doesn't automatically lift operators for nullable or option types, so the easiest way to start is to write something like this:*

```
> let lift op a b =
   match a, b with
   | Some(av), Some(bv) -> Some(op av bv)
   | _, _ -> None;;

val lift : ('a -> 'b -> 'c) -> 'a option -> 'b option -> 'c option

> let (+?) a b = lift (+) a b
  let (-?) a b = lift (-) a b;;

val ( +? ) : int option -> int option -> int option
val ( -? ) : int option -> int option -> int option

> (Some 10) +? (Some 32);;
val it : int option = Some 42
> (Some 10) -? None;;
val it : int option = None
```

[6] `http://stackoverflow.com/questions/946815/f-nullablet-support`
[7] `http://cs.hubfs.net/forums/thread/11296.aspx`

> *In addition, you can use a monad, e.g. something very similar to the AttemptBuilder example*[8] *on either 'option' or 'Nullable' and then use e.g.*
>
> ```
> myMonad {
>     let! a = ...
>     let! b = ...
>     return a + b
> }
> ```

4.3.2 C#

Some areas where the C# compiler is aware about nullables:

- null comparison, checks `HasValue` property.
- ternary operator `??` under the hood: testing `HasValue` and conditionally calling `GetValueOrDefault`.
- the following two lines are equivalent:

```
int? a = null;
Nullable<int> b = new Nullable<int>();
```

4.3.3 CLR

> *The CLR knows about nullable types too - it makes sure that if you box the null value of a nullable type, you end up with a null reference. (And likewise you can unbox a null reference to the null value of a nullable type.)*

Getting the design right for nullable support took a few iterations: `http://blogs.msdn.com/b/somasegar/archive/2005/08/11/450640.aspx`

> *As several of you pointed out, the `Nullable` type worked well only in strongly-typed scenarios. Once an instance of the type was boxed (by casting to the base `Object` type), it became a boxed value type, and no matter what its original `null` state claimed, the boxed value-type was never `null`.*
>
> ```
> int? x = null;
> object y = x;
> if (y == null) { // oops, it is not null?
>   ...
> }
> ```
>
> *It also became increasingly difficult to tell whether a variable used in a generic type or method was ever null.*
>
> ```
> void Foo<T>(T t) {
>    if (t == null) { // never true if T is a Nullable<S>?
>    }
> }
> ```

[8] `http://blogs.msdn.com/dsyme/archive/2007/09/22/some-details-on-f-computation-expressions-aka-monadic-or-wor aspx`

Clearly this had to change . . .

The outcome is that the `Nullable` *type is now* ***a new basic runtime intrinsic. It is still declared as a generic value-type, yet the runtime treats it special****. One of the foremost changes is that boxing now honors the null state. A* `Nullabe` *int now boxes to become not a "boxed Nullable int" but a "boxed int" (or a "null reference" as the null state may indicate.) Likewise, it is now possible to unbox any kind of "boxed valuetype" into its "nullable type" equivalent.*

```
int x = 10;
object y = x;
int? z = (int?) y; // unbox into a Nullable<int>
```

Together, these changes allow you to mix and match `Nullable` *types with boxed types in a variety of loosely typed API's such as reflection. Each becomes an alternative, interchangeable representation of the other.*

The C# language introduced additional behaviors that make the difference between the `Nullable` *type and reference types even more seamless. For example, since boxing now removes the wrapper, boxing instead the enclosed type, other kinds of coercions that also implied boxing became interesting. It is now possible to coerce a Nullable type to an interface implemented by the enclosed type.*

```
int? x = 0;
IComparable<int> ic = x; // implicit coercion
```

TODO: Nullables behave on the evaluation stack like the named wrapper classes (`java.lang.Long`, etc.) of Java. On the other hand, their contents are accessed not through boxing/unboxing but using `HasValue`, `get_Value`, and `GetValueOrDefault`.

5 Sidenotes

- Comments on CLR verification, `http://higherlogics.blogspot.com/2010/05/cil-verification-and-safety.html`
- `http://higherlogics.blogspot.com/search/label/CIL`
- peverify reacting differently for `unbox - ldobj` on v1.1 vs. v2.0, `http://objectmix.com/dotnet/97673-peverify-reacting-differently-unbox-ldobj-v1-1-v2-0 html`

5.1 FYI: Unsigned integrals in other .NET languages

F# supports all CIL integral types, including the non-portable native layouts. For example [1, §3.8], literals of each integral type can be:

```
token sbyte     = xint 'y'        -- e.g., 34y
token byte      = xint 'uy'       -- e.g., 34uy
```

```
token int16     = xint 's'          -- e.g., 34s
token uint16    = xint 'us'         -- e.g., 34us

token int32     = xint 'l'          -- e.g., 34l
token uint32    = xint 'ul'         -- e.g., 34ul
                | xint 'u'          -- e.g., 34u

token nativeint = xint 'n'          -- e.g., 34n
token unativeint = xint 'un'        -- e.g., 34un

token int64     = xint 'L'          -- e.g., 34L
token uint64    = xint 'UL'         -- e.g., 34UL
                | xint 'uL'         -- e.g., 34uL
```

5.2 FYI: `AddressOf` and `byref<ty>` in F#

Quoting from [1, §6.5.5] (*The `AddressOf` Operators*):

> *Under default definitions, expressions of the forms*
>
> ```
>  &expr
> &&expr
> ```
>
> *are address-of expressions, called byref-address-of expression and nativeptr-address-of expression respectively. These take the address of a mutable local variable, byref-valued argument, field, array element of static mutable global variable.*
>
> *For `&expr` and `&&expr`, the initial type of the overall expression must be of the form `byref<ty>` and `nativeptr<ty>` respectively, and the expression expr is checked with initial type `ty`. The overall expression is elaborated recursively by taking the address of the elaborated form of expr, written `AddressOf(expr, DefinitelyMutates)`, defined in §6.10.3. Note: Use of these operators may result in unverifiable or invalid CIL code , and a warning or error will typically be given if this is possible. . . . Addresses generated by the `&&` operator must not be passed to functions that are in tailcall position. This is not checked by the F# compiler. . . .*
>
> *Note: The rules in this section apply to uses of the prefix operators*
>
> ```
> Microsoft.FSharp.Core.LanguagePrimitives.IntrinsicOperators.(~&)
> Microsoft.FSharp.Core.LanguagePrimitives.IntrinsicOperators.(~&&)
> ```
>
> *defined in the F# core library when applied to one argument. Other uses of these operators are not permitted.*

References

[1] The F# 2.0 Language Specification (April 2010). `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html`.

[2] Microsoft Corporation. C# version 3.0 language specification, 2007. `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

Listing 2: Sec. 4

```
.method private hidebysig static void NullablesSample() cil managed
{
/*- rather than displaying the csc generated temp locals CS$0$0000, CS$0$0001, and CS$0$0002
    we use C0, C1, C2 instead. */
  // Code size       84 (0x54)
  .maxstack 3
  .locals init ([0] valuetype [mscorlib]System.Nullable`1<int32> ni1,
           [1] valuetype [mscorlib]System.Nullable`1<int32> ni2,
           [2] int32 nres,
           [3] valuetype [mscorlib]System.Nullable`1<int32> C0,
           [4] valuetype [mscorlib]System.Nullable`1<int32> C1,
           [5] valuetype [mscorlib]System.Nullable`1<int32> C2)
  IL_0000: nop
  IL_0001: ldloca.s  ni1
  IL_0003: initobj   valuetype [mscorlib]System.Nullable`1<int32>
  IL_0009: ldloca.s  ni2
  IL_000b: ldc.i4.s  10
  IL_000d: call      instance void valuetype [mscorlib]System.Nullable`1<int32>::.ctor(!0)
  IL_0012: nop
  IL_0013: ldloc.0
  IL_0014: stloc.3
  IL_0015: ldloc.1
  IL_0016: stloc.s   C1
  IL_0018: ldloca.s  C0
  IL_001a: call      instance bool valuetype [mscorlib]System.Nullable`1<int32>::get_HasValue()
  IL_001f: ldloca.s  C1
  IL_0021: call      instance bool valuetype [mscorlib]System.Nullable`1<int32>::get_HasValue()
  IL_0026: and
  IL_0027: brtrue.s  IL_0035
  IL_0029: ldloca.s  C2
  IL_002b: initobj   valuetype [mscorlib]System.Nullable`1<int32>
  IL_0031: ldloc.s   C2
  IL_0033: br.s      IL_0049
  IL_0035: ldloca.s  C0
  IL_0037: call      instance !0 valuetype [mscorlib]System.Nullable`1<int32>::GetValueOrDefault()
  IL_003c: ldloca.s  C1
  IL_003e: call      instance !0 valuetype [mscorlib]System.Nullable`1<int32>::GetValueOrDefault()
  IL_0043: add
  IL_0044: newobj    instance void valuetype [mscorlib]System.Nullable`1<int32>::.ctor(!0)
  IL_0049: stloc.s   C2
  IL_004b: ldloca.s  C2
  IL_004d: call      instance !0 valuetype [mscorlib]System.Nullable`1<int32>::get_Value()
  IL_0052: stloc.2
  IL_0053: ret
} // end of method XDemo::NullablesSample
```