

LinksPlatform's Platform.Reflection Class Library

1.1 ./csharp/Platform.Reflection/AssemblyExtensions.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Reflection;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Collections.Lists;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class AssemblyExtensions
13     {
14         private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
15             ↳ ConcurrentDictionary<Assembly, Type[]>();
16
17         /// <remarks>
18         /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
19         /// </remarks>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static Type[] GetLoadableTypes(this Assembly assembly)
22         {
23             Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
24             try
25             {
26                 return assembly.GetTypes();
27             }
28             catch (ReflectionTypeLoadException e)
29             {
30                 return e.Types.ToArray(t => t != null);
31             }
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
36             ↳ _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
37     }
38 }
```

1.2 ./csharp/Platform.Reflection/DelegateHelpers.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
6 using Platform.Exceptions;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class DelegateHelpers
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
16             ↳ typeMemberMethod)
17             where TDelegate : Delegate
18         {
19             var @delegate = default(TDelegate);
20             try
21             {
22                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
23                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
24             }
25             catch (Exception exception)
26             {
27                 exception.Ignore();
28             }
29             return @delegate;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
34             ↳ TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
38             ↳ typeMemberMethod)
39         {
40             // ...
41         }
42     }
43 }
```

```

35     where TDelegate : Delegate
36 {
37     var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
38     if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
39     {
40         @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
41     }
42     return @delegate;
43 }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
    ↳ : Delegate => Compile<TDelegate>(emitCode, false);
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
50 {
51     var delegateType = typeof(TDelegate);
52     delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
        ↳ parameterTypes);
53     var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
54     emitCode(dynamicMethod.GetILGenerator());
55     return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
60 {
61     AssemblyName assemblyName = new AssemblyName(GetNewName());
62     var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
        ↳ AssemblyBuilderAccess.Run);
63     var module = assembly.DefineDynamicModule(GetNewName());
64     var type = module.DefineType(GetNewName());
65     var methodName = GetNewName();
66     type.EmitStaticMethod<TDelegate>(methodName, emitCode);
67     var typeInfo = type.CreateTypeInfo();
68     return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
        ↳ gate));
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 private static string GetNewName() => Guid.NewGuid().ToString("N");
73 }
74 }

```

1.3 ./csharp/Platform.Reflection/DynamicExtensions.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class DynamicExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static bool HasProperty(this object @object, string propertyName)
12         {
13             var type = @object.GetType();
14             if (type is IDictionary<string, object> dictionary)
15             {
16                 return dictionary.ContainsKey(propertyName);
17             }
18             return type.GetProperty(propertyName) != null;
19         }
20     }
21 }

```

1.4 ./csharp/Platform.Reflection/EnsureExtensions.cs

```

1 using System;
2 using System.Diagnostics;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Exceptions.ExtensionRoots;
6
7 #pragma warning disable IDE0060 // Remove unused parameter
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9

```

```

10 namespace Platform.Reflection
11 {
12     public static class EnsureExtensions
13     {
14         #region Always
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root,
18         ↪ Func<string> messageBuilder)
19         {
20             if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
21             ↪ NumericType<T>.IsFloatPoint)
22             {
23                 throw new NotSupportedException(messageBuilder());
24             }
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
29         ↪ message)
30         {
31             string messageBuilder() => message;
32             IsUnsignedInteger<T>(root, messageBuilder());
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
37         ↪ IsUnsignedInteger<T>(root, (string)null);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
41         ↪ messageBuilder)
42         {
43             if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
44             ↪ NumericType<T>.IsFloatPoint)
45             {
46                 throw new NotSupportedException(messageBuilder());
47             }
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
52         ↪ message)
53         {
54             string messageBuilder() => message;
55             IsSignedInteger<T>(root, messageBuilder());
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
60         ↪ IsSignedInteger<T>(root, (string)null);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
64         ↪ messageBuilder)
65         {
66             if (!NumericType<T>.IsSigned)
67             {
68                 throw new NotSupportedException(messageBuilder());
69             }
70         }
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
74         {
75             string messageBuilder() => message;
76             IsSigned<T>(root, messageBuilder());
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
81         ↪ (string)null);
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
85         ↪ messageBuilder)
86         {
87             if (!NumericType<T>.IsNumeric)

```

```

    {
        throw new NotSupportedException(messageBuilder());
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
{
    string messageBuilder() => message;
    IsNumeric<T>(root, messageBuilder());
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↪ IsNumeric<T>(root, (string)null);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
    ↪ messageBuilder)
{
    if (!NumericType<T>.CanBeNumeric)
    {
        throw new NotSupportedException(messageBuilder());
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
{
    string messageBuilder() => message;
    CanBeNumeric<T>(root, messageBuilder());
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
    ↪ CanBeNumeric<T>(root, (string)null);

#endregion

#region OnDebug

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
    ↪ Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↪ message) => Ensure.Always.IsUnsignedInteger<T>(message);

[Conditional("DEBUG")]
public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsUnsignedInteger<T>();

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↪ messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
    ↪ message) => Ensure.Always.IsSignedInteger<T>(message);

[Conditional("DEBUG")]
public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsSignedInteger<T>();

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
    ↪ messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
    ↪ Ensure.Always.IsSigned<T>(message);

[Conditional("DEBUG")]
public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
    ↪ Ensure.Always.IsSigned<T>();

[Conditional("DEBUG")]

```

```

143     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
        ↳ messageBuilder) => Ensure.Always.IsNumeric<T>(messageBuilder);
144
145     [Conditional("DEBUG")]
146     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root, string message) =>
        ↳ Ensure.Always.IsNumeric<T>(message);
147
148     [Conditional("DEBUG")]
149     public static void IsNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
        ↳ Ensure.Always.IsNumeric<T>();
150
151     [Conditional("DEBUG")]
152     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, Func<string>
        ↳ messageBuilder) => Ensure.Always.CanBeNumeric<T>(messageBuilder);
153
154     [Conditional("DEBUG")]
155     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root, string message)
        ↳ => Ensure.Always.CanBeNumeric<T>(message);
156
157     [Conditional("DEBUG")]
158     public static void CanBeNumeric<T>(this EnsureOnDebugExtensionRoot root) =>
        ↳ Ensure.Always.CanBeNumeric<T>();
159
160     #endregion
161 }
162 }

```

1.5 ./csharp/Platform.Reflection/FieldInfoExtensions.cs

```

1 using System.Reflection;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class FieldInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this FieldInfo fieldInfo) =>
            ↳ (T)fieldInfo.GetValue(null);
12     }
13 }

```

1.6 ./csharp/Platform.Reflection/ILGeneratorExtensions.cs

```

1 using System;
2 using System.Linq;
3 using System.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Reflection
10 {
11     public static class ILGeneratorExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void Throw<T>(this ILGenerator generator) =>
            ↳ generator.ThrowException(typeof(T));
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator) =>
            ↳ UncheckedConvert<TSource, TTarget>(generator, extendSign: false);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static void UncheckedConvert<TSource, TTarget>(this ILGenerator generator, bool
            ↳ extendSign)
21         {
22             var sourceType = typeof(TSource);
23             var targetType = typeof(TTarget);
24             if (sourceType == targetType)
25             {
26                 return;
27             }
28             if (extendSign)
29             {
30                 if (sourceType == typeof(byte))
31                 {

```

```

32         generator.Emit(OpCodes.Conv_I1);
33     }
34     if (sourceType == typeof(ushort) || sourceType == typeof(char))
35     {
36         generator.Emit(OpCodes.Conv_I2);
37     }
38 }
39 if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
40 {
41     generator.ConvertToInteger<TSource>(targetType, extendSign: false);
42 }
43 else
44 {
45     generator.ConvertToInteger<TSource>(targetType, extendSign);
46 }
47 if (targetType == typeof(float))
48 {
49     if (NumericType<TSource>.IsSigned)
50     {
51         generator.Emit(OpCodes.Conv_R4);
52     }
53     else
54     {
55         generator.Emit(OpCodes.Conv_R_Un);
56     }
57 }
58 else if (targetType == typeof(double))
59 {
60     generator.Emit(OpCodes.Conv_R8);
61 }
62 else if (targetType == typeof(bool))
63 {
64     generator.ConvertToBoolean<TSource>();
65 }
66 }
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 private static void ConvertToBoolean<TSource>(this ILGenerator generator)
70 {
71     generator.LoadConstant<TSource>(default);
72     var sourceType = typeof(TSource);
73     if (sourceType == typeof(float) || sourceType == typeof(double))
74     {
75         generator.Emit(OpCodes.Ceq);
76         // Inversion of the first Ceq instruction
77         generator.LoadConstant<int>(0);
78         generator.Emit(OpCodes.Ceq);
79     }
80     else
81     {
82         generator.Emit(OpCodes.Cgt_Un);
83     }
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
88 ↪ targetType, bool extendSign)
89 {
90     if (targetType == typeof(sbyte))
91     {
92         generator.Emit(OpCodes.Conv_I1);
93     }
94     else if (targetType == typeof(byte))
95     {
96         generator.Emit(OpCodes.Conv_U1);
97     }
98     else if (targetType == typeof(short))
99     {
100         generator.Emit(OpCodes.Conv_I2);
101     }
102     else if (targetType == typeof(ushort) || targetType == typeof(char))
103     {
104         var sourceType = typeof(TSource);
105         if (sourceType != typeof(ushort) && sourceType != typeof(char))
106         {
107             generator.Emit(OpCodes.Conv_U2);
108         }
109     }
110 }

```

```

109     else if (targetType == typeof(int))
110     {
111         generator.Emit(OpCodes.Conv_I4);
112     }
113     else if (targetType == typeof(uint))
114     {
115         generator.Emit(OpCodes.Conv_U4);
116     }
117     else if (targetType == typeof(long) || targetType == typeof(ulong))
118     {
119         if (NumericType<TSource>.IsSigned || extendSign)
120         {
121             generator.Emit(OpCodes.Conv_I8);
122         }
123         else
124         {
125             generator.Emit(OpCodes.Conv_U8);
126         }
127     }
128 }
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
132 {
133     var sourceType = typeof(TSource);
134     var targetType = typeof(TTarget);
135     if (sourceType == targetType)
136     {
137         return;
138     }
139     if (targetType == typeof(short))
140     {
141         if (NumericType<TSource>.IsSigned)
142         {
143             generator.Emit(OpCodes.Conv_Ovf_I2);
144         }
145         else
146         {
147             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
148         }
149     }
150     else if (targetType == typeof(ushort) || targetType == typeof(char))
151     {
152         if (sourceType != typeof(ushort) && sourceType != typeof(char))
153         {
154             if (NumericType<TSource>.IsSigned)
155             {
156                 generator.Emit(OpCodes.Conv_Ovf_U2);
157             }
158             else
159             {
160                 generator.Emit(OpCodes.Conv_Ovf_U2_Un);
161             }
162         }
163     }
164     else if (targetType == typeof(sbyte))
165     {
166         if (NumericType<TSource>.IsSigned)
167         {
168             generator.Emit(OpCodes.Conv_Ovf_I1);
169         }
170         else
171         {
172             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
173         }
174     }
175     else if (targetType == typeof(byte))
176     {
177         if (NumericType<TSource>.IsSigned)
178         {
179             generator.Emit(OpCodes.Conv_Ovf_U1);
180         }
181         else
182         {
183             generator.Emit(OpCodes.Conv_Ovf_U1_Un);
184         }
185     }
186     else if (targetType == typeof(int))

```

```

187     {
188         if (NumericType<TSource>.IsSigned)
189         {
190             generator.Emit(OpCodes.Conv_Ovf_I4);
191         }
192         else
193         {
194             generator.Emit(OpCodes.Conv_Ovf_I4_Un);
195         }
196     }
197     else if (targetType == typeof(uint))
198     {
199         if (NumericType<TSource>.IsSigned)
200         {
201             generator.Emit(OpCodes.Conv_Ovf_U4);
202         }
203         else
204         {
205             generator.Emit(OpCodes.Conv_Ovf_U4_Un);
206         }
207     }
208     else if (targetType == typeof(long))
209     {
210         if (NumericType<TSource>.IsSigned)
211         {
212             generator.Emit(OpCodes.Conv_Ovf_I8);
213         }
214         else
215         {
216             generator.Emit(OpCodes.Conv_Ovf_I8_Un);
217         }
218     }
219     else if (targetType == typeof(ulong))
220     {
221         if (NumericType<TSource>.IsSigned)
222         {
223             generator.Emit(OpCodes.Conv_Ovf_U8);
224         }
225         else
226         {
227             generator.Emit(OpCodes.Conv_Ovf_U8_Un);
228         }
229     }
230     else if (targetType == typeof(float))
231     {
232         if (NumericType<TSource>.IsSigned)
233         {
234             generator.Emit(OpCodes.Conv_R4);
235         }
236         else
237         {
238             generator.Emit(OpCodes.Conv_R_Un);
239         }
240     }
241     else if (targetType == typeof(double))
242     {
243         generator.Emit(OpCodes.Conv_R8);
244     }
245     else if (targetType == typeof(bool))
246     {
247         generator.ConvertToBoolean<TSource>();
248     }
249     else
250     {
251         throw new NotSupportedException();
252     }
253 }
254
255 [MethodImpl(MethodImplOptions.AggressiveInlining)]
256 public static void LoadConstant(this ILGenerator generator, bool value) =>
257     ↪ generator.LoadConstant(value ? 1 : 0);
258
259 [MethodImpl(MethodImplOptions.AggressiveInlining)]
260 public static void LoadConstant(this ILGenerator generator, float value) =>
261     ↪ generator.Emit(OpCodes.Ldc_R4, value);
262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

262 public static void LoadConstant(this ILGenerator generator, double value) =>
263     ↪ generator.Emit(OpCodes.Ldc_R8, value);
264
265 [MethodImpl(MethodImplOptions.AggressiveInlining)]
266 public static void LoadConstant(this ILGenerator generator, ulong value) =>
267     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
268
269 [MethodImpl(MethodImplOptions.AggressiveInlining)]
270 public static void LoadConstant(this ILGenerator generator, long value) =>
271     ↪ generator.Emit(OpCodes.Ldc_I8, value);
272
273 [MethodImpl(MethodImplOptions.AggressiveInlining)]
274 public static void LoadConstant(this ILGenerator generator, uint value)
275 {
276     switch (value)
277     {
278         case uint.MaxValue:
279             generator.Emit(OpCodes.Ldc_I4_M1);
280             return;
281         case 0:
282             generator.Emit(OpCodes.Ldc_I4_0);
283             return;
284         case 1:
285             generator.Emit(OpCodes.Ldc_I4_1);
286             return;
287         case 2:
288             generator.Emit(OpCodes.Ldc_I4_2);
289             return;
290         case 3:
291             generator.Emit(OpCodes.Ldc_I4_3);
292             return;
293         case 4:
294             generator.Emit(OpCodes.Ldc_I4_4);
295             return;
296         case 5:
297             generator.Emit(OpCodes.Ldc_I4_5);
298             return;
299         case 6:
300             generator.Emit(OpCodes.Ldc_I4_6);
301             return;
302         case 7:
303             generator.Emit(OpCodes.Ldc_I4_7);
304             return;
305         case 8:
306             generator.Emit(OpCodes.Ldc_I4_8);
307             return;
308         default:
309             if (value <= sbyte.MaxValue)
310             {
311                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
312             }
313             else
314             {
315                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
316             }
317             return;
318     }
319 }
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 public static void LoadConstant(this ILGenerator generator, int value)
323 {
324     switch (value)
325     {
326         case -1:
327             generator.Emit(OpCodes.Ldc_I4_M1);
328             return;
329         case 0:
330             generator.Emit(OpCodes.Ldc_I4_0);
331             return;
332         case 1:
333             generator.Emit(OpCodes.Ldc_I4_1);
334             return;
335         case 2:
336             generator.Emit(OpCodes.Ldc_I4_2);
337             return;
338         case 3:
339             generator.Emit(OpCodes.Ldc_I4_3);
340             return;
341         case 4:
342             generator.Emit(OpCodes.Ldc_I4_4);
343             return;
344         case 5:
345             generator.Emit(OpCodes.Ldc_I4_5);
346             return;
347         case 6:
348             generator.Emit(OpCodes.Ldc_I4_6);
349             return;
350         case 7:
351             generator.Emit(OpCodes.Ldc_I4_7);
352             return;
353         case 8:
354             generator.Emit(OpCodes.Ldc_I4_8);
355             return;
356         default:
357             if (value <= sbyte.MaxValue)
358             {
359                 generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
360             }
361             else
362             {
363                 generator.Emit(OpCodes.Ldc_I4, unchecked((int)value));
364             }
365             return;
366     }
367 }

```

```

340         return;
341     case 5:
342         generator.Emit(OpCodes.Ldc_I4_5);
343         return;
344     case 6:
345         generator.Emit(OpCodes.Ldc_I4_6);
346         return;
347     case 7:
348         generator.Emit(OpCodes.Ldc_I4_7);
349         return;
350     case 8:
351         generator.Emit(OpCodes.Ldc_I4_8);
352         return;
353     default:
354         if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
355         {
356             generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
357         }
358         else
359         {
360             generator.Emit(OpCodes.Ldc_I4, value);
361         }
362         return;
363     }
364 }
365
366 [MethodImpl(MethodImplOptions.AggressiveInlining)]
367 public static void LoadConstant(this ILGenerator generator, short value) =>
368     ↪ generator.LoadConstant((int)value);
369
370 [MethodImpl(MethodImplOptions.AggressiveInlining)]
371 public static void LoadConstant(this ILGenerator generator, ushort value) =>
372     ↪ generator.LoadConstant((int)value);
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 public static void LoadConstant(this ILGenerator generator, sbyte value) =>
376     ↪ generator.LoadConstant((int)value);
377
378 [MethodImpl(MethodImplOptions.AggressiveInlining)]
379 public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
380     ↪ LoadConstantOne(generator, typeof(TConstant));
381
382 [MethodImpl(MethodImplOptions.AggressiveInlining)]
383 public static void LoadConstantOne(this ILGenerator generator, Type constantType)
384 {
385     if (constantType == typeof(float))
386     {
387         generator.LoadConstant(1F);
388     }
389     else if (constantType == typeof(double))
390     {
391         generator.LoadConstant(1D);
392     }
393     else if (constantType == typeof(long))
394     {
395         generator.LoadConstant(1L);
396     }
397     else if (constantType == typeof(ulong))
398     {
399         generator.LoadConstant(1UL);
400     }
401     else if (constantType == typeof(int))
402     {
403         generator.LoadConstant(1);
404     }
405     else if (constantType == typeof(uint))
406     {
407         generator.LoadConstant(1U);
408     }
409     else if (constantType == typeof(short))
410     {
411         generator.LoadConstant((short)1);
412     }
413     else if (constantType == typeof(ushort))
414     {

```

```

414         generator.LoadConstant((ushort)1);
415     }
416     else if (constantType == typeof(sbyte))
417     {
418         generator.LoadConstant((sbyte)1);
419     }
420     else if (constantType == typeof(byte))
421     {
422         generator.LoadConstant((byte)1);
423     }
424     else
425     {
426         throw new NotSupportedException();
427     }
428 }
429
430 [MethodImpl(MethodImplOptions.AggressiveInlining)]
431 public static void LoadConstant<TConstant>(this ILGenerator generator, TConstant
    ↪ constantValue) => LoadConstant(generator, typeof(TConstant), constantValue);
432
433 [MethodImpl(MethodImplOptions.AggressiveInlining)]
434 public static void LoadConstant(this ILGenerator generator, Type constantType, object
    ↪ constantValue)
435 {
436     constantValue = Convert.ChangeType(constantValue, constantType);
437     if (constantType == typeof(float))
438     {
439         generator.LoadConstant((float)constantValue);
440     }
441     else if (constantType == typeof(double))
442     {
443         generator.LoadConstant((double)constantValue);
444     }
445     else if (constantType == typeof(long))
446     {
447         generator.LoadConstant((long)constantValue);
448     }
449     else if (constantType == typeof(ulong))
450     {
451         generator.LoadConstant((ulong)constantValue);
452     }
453     else if (constantType == typeof(int))
454     {
455         generator.LoadConstant((int)constantValue);
456     }
457     else if (constantType == typeof(uint))
458     {
459         generator.LoadConstant((uint)constantValue);
460     }
461     else if (constantType == typeof(short))
462     {
463         generator.LoadConstant((short)constantValue);
464     }
465     else if (constantType == typeof(ushort))
466     {
467         generator.LoadConstant((ushort)constantValue);
468     }
469     else if (constantType == typeof(sbyte))
470     {
471         generator.LoadConstant((sbyte)constantValue);
472     }
473     else if (constantType == typeof(byte))
474     {
475         generator.LoadConstant((byte)constantValue);
476     }
477     else
478     {
479         throw new NotSupportedException();
480     }
481 }
482
483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
484 public static void Increment<TValue>(this ILGenerator generator) =>
    ↪ generator.Increment(typeof(TValue));
485
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 public static void Decrement<TValue>(this ILGenerator generator) =>
    ↪ generator.Decrement(typeof(TValue));

```

```

488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public static void Increment(this ILGenerator generator, Type valueType)
490 {
491     generator.LoadConstantOne(valueType);
492     generator.Add();
493 }
494
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static void Add(this ILGenerator generator) => generator.Emit(OpCodes.Add);
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 public static void Decrement(this ILGenerator generator, Type valueType)
500 {
501     generator.LoadConstantOne(valueType);
502     generator.Subtract();
503 }
504
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 public static void Subtract(this ILGenerator generator) => generator.Emit(OpCodes.Sub);
507
508 [MethodImpl(MethodImplOptions.AggressiveInlining)]
509 public static void Negate(this ILGenerator generator) => generator.Emit(OpCodes.Neg);
510
511 [MethodImpl(MethodImplOptions.AggressiveInlining)]
512 public static void And(this ILGenerator generator) => generator.Emit(OpCodes.And);
513
514 [MethodImpl(MethodImplOptions.AggressiveInlining)]
515 public static void Or(this ILGenerator generator) => generator.Emit(OpCodes.Or);
516
517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
518 public static void Not(this ILGenerator generator) => generator.Emit(OpCodes.Not);
519
520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
521 public static void ShiftLeft(this ILGenerator generator) => generator.Emit(OpCodes.Shl);
522
523 [MethodImpl(MethodImplOptions.AggressiveInlining)]
524 public static void ShiftRight(this ILGenerator generator) => generator.Emit(OpCodes.Shr);
525
526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
527 public static void LoadArgument(this ILGenerator generator, int argumentIndex)
528 {
529     switch (argumentIndex)
530     {
531         case 0:
532             generator.Emit(OpCodes.Ldarg_0);
533             break;
534         case 1:
535             generator.Emit(OpCodes.Ldarg_1);
536             break;
537         case 2:
538             generator.Emit(OpCodes.Ldarg_2);
539             break;
540         case 3:
541             generator.Emit(OpCodes.Ldarg_3);
542             break;
543         default:
544             generator.Emit(OpCodes.Ldarg, argumentIndex);
545             break;
546     }
547 }
548
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 public static void LoadArguments(this ILGenerator generator, params int[]
551     ↪ argumentIndices)
552 {
553     for (var i = 0; i < argumentIndices.Length; i++)
554     {
555         generator.LoadArgument(argumentIndices[i]);
556     }
557 }
558
559 [MethodImpl(MethodImplOptions.AggressiveInlining)]
560 public static void StoreArgument(this ILGenerator generator, int argumentIndex) =>
561     ↪ generator.Emit(OpCodes.Starg, argumentIndex);
562
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 public static void CompareGreaterThan(this ILGenerator generator) =>
565     ↪ generator.Emit(OpCodes.Cgt);

```

```

564 [MethodImpl(MethodImplOptions.AggressiveInlining)]
565 public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
566     ↪ generator.Emit(OpCodes.Cgt_Un);

567 [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
569 {
570     if (isSigned)
571     {
572         generator.CompareGreaterThan();
573     }
574     else
575     {
576         generator.UnsignedCompareGreaterThan();
577     }
578 }
579 }

580 [MethodImpl(MethodImplOptions.AggressiveInlining)]
581 public static void CompareLessThan(this ILGenerator generator) =>
582     ↪ generator.Emit(OpCodes.Clt);

583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
585     ↪ generator.Emit(OpCodes.Clt_Un);

586 [MethodImpl(MethodImplOptions.AggressiveInlining)]
587 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
588 {
589     if (isSigned)
590     {
591         generator.CompareLessThan();
592     }
593     else
594     {
595         generator.UnsignedCompareLessThan();
596     }
597 }
598 }

599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
601     ↪ generator.Emit(OpCodes.Bge, label);

602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
604     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);

605 [MethodImpl(MethodImplOptions.AggressiveInlining)]
606 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
607     ↪ Label label)
608 {
609     if (isSigned)
610     {
611         generator.BranchIfGreaterOrEqual(label);
612     }
613     else
614     {
615         generator.UnsignedBranchIfGreaterOrEqual(label);
616     }
617 }

618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
619 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
620     ↪ generator.Emit(OpCodes.Ble, label);

621 [MethodImpl(MethodImplOptions.AggressiveInlining)]
622 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
623     ↪ => generator.Emit(OpCodes.Ble_Un, label);

624 [MethodImpl(MethodImplOptions.AggressiveInlining)]
625 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
626     ↪ label)
627 {
628     if (isSigned)
629     {
630         generator.BranchIfLessOrEqual(label);
631     }
632     else

```

```

633     {
634         generator.UnsignedBranchIfLessOrEqual(label);
635     }
636 }
637
638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
639 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 public static void Box(this ILGenerator generator, Type boxedType) =>
643     ↪ generator.Emit(OpCodes.Box, boxedType);
644
645 [MethodImpl(MethodImplOptions.AggressiveInlining)]
646 public static void Call(this ILGenerator generator, MethodInfo method) =>
647     ↪ generator.Emit(OpCodes.Call, method);
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static void Unbox<TUnbox>(this ILGenerator generator) =>
654     ↪ generator.Unbox(typeof(TUnbox));
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
658     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
659
660 [MethodImpl(MethodImplOptions.AggressiveInlining)]
661 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
662     ↪ generator.UnboxValue(typeof(TUnbox));
663
664 [MethodImpl(MethodImplOptions.AggressiveInlining)]
665 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
666     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
667
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
670     ↪ generator.DeclareLocal(typeof(T));
671
672 [MethodImpl(MethodImplOptions.AggressiveInlining)]
673 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
674     ↪ generator.Emit(OpCodes.Ldloc, local);
675
676 [MethodImpl(MethodImplOptions.AggressiveInlining)]
677 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
678     ↪ generator.Emit(OpCodes.Stloc, local);
679
680 [MethodImpl(MethodImplOptions.AggressiveInlining)]
681 public static void NewObject(this ILGenerator generator, Type type, params Type[]
682     ↪ parameterTypes)
683 {
684     var allConstructors = type.GetConstructors(BindingFlags.Public |
685         ↪ BindingFlags.NonPublic | BindingFlags.Instance
686         ↪ BindingFlags.CreateInstance
687         ↪ );
688     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
689         ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
690         ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
691     if (constructor == null)
692     {
693         throw new InvalidOperationException("Type " + type + " must have a constructor
694             ↪ that matches parameters [" + string.Join(", ",
695             ↪ parameterTypes.AsEnumerable()) + "]");
696     }
697     generator.NewObject(constructor);
698 }
699
700 [MethodImpl(MethodImplOptions.AggressiveInlining)]
701 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
702     ↪ generator.Emit(OpCodes.Newobj, constructor);
703
704 [MethodImpl(MethodImplOptions.AggressiveInlining)]
705 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
706     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
707
708 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

694 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
    ↪ false, byte? unaligned = null)
695 {
696     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
697     {
698         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
699     }
700     if (isVolatile)
701     {
702         generator.Emit(OpCodes.Volatile);
703     }
704     if (unaligned.HasValue)
705     {
706         generator.Emit(OpCodes.Unaligned, unaligned.Value);
707     }
708     if (type.IsPointer)
709     {
710         generator.Emit(OpCodes.Ldind_I);
711     }
712     else if (!type.IsValueType)
713     {
714         generator.Emit(OpCodes.Ldind_Ref);
715     }
716     else if (type == typeof(sbyte))
717     {
718         generator.Emit(OpCodes.Ldind_I1);
719     }
720     else if (type == typeof(bool))
721     {
722         generator.Emit(OpCodes.Ldind_I1);
723     }
724     else if (type == typeof(byte))
725     {
726         generator.Emit(OpCodes.Ldind_U1);
727     }
728     else if (type == typeof(short))
729     {
730         generator.Emit(OpCodes.Ldind_I2);
731     }
732     else if (type == typeof(ushort))
733     {
734         generator.Emit(OpCodes.Ldind_U2);
735     }
736     else if (type == typeof(char))
737     {
738         generator.Emit(OpCodes.Ldind_U2);
739     }
740     else if (type == typeof(int))
741     {
742         generator.Emit(OpCodes.Ldind_I4);
743     }
744     else if (type == typeof(uint))
745     {
746         generator.Emit(OpCodes.Ldind_U4);
747     }
748     else if (type == typeof(long) || type == typeof(ulong))
749     {
750         generator.Emit(OpCodes.Ldind_I8);
751     }
752     else if (type == typeof(float))
753     {
754         generator.Emit(OpCodes.Ldind_R4);
755     }
756     else if (type == typeof(double))
757     {
758         generator.Emit(OpCodes.Ldind_R8);
759     }
760     else
761     {
762         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
    ↪ ", LoadObject may be more appropriate");
763     }
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
    ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
768

```

```

769 [MethodImpl(MethodImplOptions.AggressiveInlining)]
770 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
↪ = false, byte? unaligned = null)
771 {
772     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
773     {
774         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
775     }
776     if (isVolatile)
777     {
778         generator.Emit(OpCodes.Volatile);
779     }
780     if (unaligned.HasValue)
781     {
782         generator.Emit(OpCodes.Unaligned, unaligned.Value);
783     }
784     if (type.IsPointer)
785     {
786         generator.Emit(OpCodes.Stind_I);
787     }
788     else if (!type.IsValueType)
789     {
790         generator.Emit(OpCodes.Stind_Ref);
791     }
792     else if (type == typeof(sbyte) || type == typeof(byte))
793     {
794         generator.Emit(OpCodes.Stind_I1);
795     }
796     else if (type == typeof(short) || type == typeof(ushort))
797     {
798         generator.Emit(OpCodes.Stind_I2);
799     }
800     else if (type == typeof(int) || type == typeof(uint))
801     {
802         generator.Emit(OpCodes.Stind_I4);
803     }
804     else if (type == typeof(long) || type == typeof(ulong))
805     {
806         generator.Emit(OpCodes.Stind_I8);
807     }
808     else if (type == typeof(float))
809     {
810         generator.Emit(OpCodes.Stind_R4);
811     }
812     else if (type == typeof(double))
813     {
814         generator.Emit(OpCodes.Stind_R8);
815     }
816     else
817     {
818         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
↪ + ", StoreObject may be more appropriate");
819     }
820 }
821
822 [MethodImpl(MethodImplOptions.AggressiveInlining)]
823 public static void Multiply(this ILGenerator generator)
824 {
825     generator.Emit(OpCodes.Mul);
826 }
827
828 [MethodImpl(MethodImplOptions.AggressiveInlining)]
829 public static void CheckedMultiply<T>(this ILGenerator generator)
830 {
831     if (NumericType<T>.IsSigned)
832     {
833         generator.Emit(OpCodes.Mul_Ovf);
834     }
835     else
836     {
837         generator.Emit(OpCodes.Mul_Ovf_Un);
838     }
839 }
840
841 [MethodImpl(MethodImplOptions.AggressiveInlining)]
842 public static void Divide<T>(this ILGenerator generator)
843 {
844     if (NumericType<T>.IsSigned)

```



```

845         {
846             generator.Emit(OpCodes.Div);
847         }
848         else
849         {
850             generator.Emit(OpCodes.Div_Un);
851         }
852     }
853 }
854 }

```

1.7 ./csharp/Platform.Reflection/MethodInfoExtensions.cs

```

1  using System;
2  using System.Linq;
3  using System.Reflection;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public static class MethodInfoExtensions
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static byte[] GetILBytes(this MethodInfo methodInfo) =>
14             ↪ methodInfo.GetMethodBody().GetILAsByteArray();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
18             ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
19     }
20 }

```

1.8 ./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Reflection
9  {
10     public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
11         where TDelegate : Delegate
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TDelegate Create()
15         {
16             var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
17             {
18                 generator.Throw<NotSupportedException>();
19             });
20             if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
21             {
22                 throw new InvalidOperationException("Unable to compile stub delegate.");
23             }
24             return @delegate;
25         }
26     }
27 }

```

1.9 ./csharp/Platform.Reflection/NumericType.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Runtime.InteropServices;
4  using Platform.Exceptions;
5
6  // ReSharper disable AssignmentInConditionalExpression
7  // ReSharper disable BuiltInTypeReferenceStyle
8  // ReSharper disable StaticFieldInGenericType
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Reflection
12 {
13     public static class NumericType<T>
14     {
15         public static readonly Type Type;
16         public static readonly Type UnderlyingType;
17     }
18 }

```

```

17 public static readonly Type SignedVersion;
18 public static readonly Type UnsignedVersion;
19 public static readonly bool IsFloatPoint;
20 public static readonly bool IsNumeric;
21 public static readonly bool IsSigned;
22 public static readonly bool CanBeNumeric;
23 public static readonly bool IsNullable;
24 public static readonly int BytesSize;
25 public static readonly int BitsSize;
26 public static readonly T MinValue;
27 public static readonly T MaxValue;
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 static NumericType()
31 {
32     try
33     {
34         var type = typeof(T);
35         var isNullable = type.IsNullable();
36         var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
37         var canBeNumeric = underlyingType.CanBeNumeric();
38         var isNumeric = underlyingType.IsNumeric();
39         var isSigned = underlyingType.IsSigned();
40         var isFloatPoint = underlyingType.IsFloatPoint();
41         var bytesSize = Marshal.SizeOf(underlyingType);
42         var bitsSize = bytesSize * 8;
43         GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
44         GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
45             ↪ out Type unsignedVersion);
46         Type = type;
47         IsNullable = isNullable;
48         UnderlyingType = underlyingType;
49         CanBeNumeric = canBeNumeric;
50         IsNumeric = isNumeric;
51         IsSigned = isSigned;
52         IsFloatPoint = isFloatPoint;
53         BytesSize = bytesSize;
54         BitsSize = bitsSize;
55         MinValue = minValue;
56         MaxValue = maxValue;
57         SignedVersion = signedVersion;
58         UnsignedVersion = unsignedVersion;
59     }
60     catch (Exception exception)
61     {
62         exception.Ignore();
63     }
64 }
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
68 {
69     if (type == typeof(bool))
70     {
71         minValue = (T)(object>false;
72         maxValue = (T)(object>true;
73     }
74     else
75     {
76         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
77         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
78     }
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
83     ↪ signedVersion, out Type unsignedVersion)
84 {
85     if (isSigned)
86     {
87         signedVersion = type;
88         unsignedVersion = type.GetUnsignedVersionOrNull();
89     }
90     else
91     {
92         signedVersion = type.GetSignedVersionOrNull();
93         unsignedVersion = type;
94     }
95 }

```

```
95 }
```

1.10 ./csharp/Platform.Reflection/PropertyInfoExtensions.cs

```
1 using System.Reflection;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Reflection
7 {
8     public static class PropertyInfoExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T GetStaticValue<T>(this PropertyInfo fieldInfo) =>
12             (T)fieldInfo.GetValue(null);
13     }
14 }
```

1.11 ./csharp/Platform.Reflection/TypeBuilderExtensions.cs

```
1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 using System;
4 using System.Reflection;
5 using System.Reflection.Emit;
6 using System.Runtime.CompilerServices;
7
8 namespace Platform.Reflection
9 {
10     public static class TypeBuilderExtensions
11     {
12         public static readonly MethodAttributes DefaultStaticMethodAttributes =
13             MethodAttributes.Public | MethodAttributes.Static;
14         public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
15             MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
16             MethodAttributes.HideBySig;
17         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
18             MethodImplAttributes.IL | MethodImplAttributes.Managed |
19             MethodImplAttributes.AggressiveInlining;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
23             MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
24             Action<ILGenerator> emitCode)
25         {
26             typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
27                 parameterTypes);
28             EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
29                 parameterTypes, emitCode);
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
34             methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
35             parameterTypes, Action<ILGenerator> emitCode)
36         {
37             MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
38                 parameterTypes);
39             method.SetImplementationFlags(methodImplAttributes);
40             var generator = method.GetILGenerator();
41             emitCode(generator);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
46             Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
47             DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
51             methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
52             DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
53     }
54 }
```

1.12 ./csharp/Platform.Reflection/TypeExtensions.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
```

```

4 using System.Reflection;
5 using System.Runtime.CompilerServices;
6 using Platform.Collections;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Reflection
11 {
12     public static class TypeExtensions
13     {
14         static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public |
15             ↳ BindingFlags.NonPublic | BindingFlags.Static;
16         static public readonly string DefaultDelegateMethodName = "Invoke";
17
18         static private readonly HashSet<Type> _canBeNumericTypes;
19         static private readonly HashSet<Type> _isNumericTypes;
20         static private readonly HashSet<Type> _isSignedTypes;
21         static private readonly HashSet<Type> _isFloatPointTypes;
22         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
23         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         static TypeExtensions()
27         {
28             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
29                 ↳ typeof(DateTime), typeof(TimeSpan) };
30             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
31                 ↳ typeof(ulong) };
32             _canBeNumericTypes.UnionWith(_isNumericTypes);
33             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
34                 ↳ typeof(long) };
35             _canBeNumericTypes.UnionWith(_isSignedTypes);
36             _isNumericTypes.UnionWith(_isSignedTypes);
37             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
38                 ↳ typeof(float) };
39             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
40             _isNumericTypes.UnionWith(_isFloatPointTypes);
41             _isSignedTypes.UnionWith(_isFloatPointTypes);
42             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
43             {
44                 { typeof(sbyte), typeof(byte) },
45                 { typeof(short), typeof(ushort) },
46                 { typeof(int), typeof(uint) },
47                 { typeof(long), typeof(ulong) },
48             };
49             _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
50             {
51                 { typeof(byte), typeof(sbyte) },
52                 { typeof(ushort), typeof(short) },
53                 { typeof(uint), typeof(int) },
54                 { typeof(ulong), typeof(long) },
55             };
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public static T GetStaticFieldValue<T>(this Type type, string name) =>
63             ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public static T GetStaticPropertyValue<T>(this Type type, string name) =>
67             ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
71             ↳ genericParameterTypes, Type[] argumentTypes)
72         {
73             var methods = from m in type.GetMethods()
74                 where m.Name == name
75                     && m.IsGenericMethodDefinition
76                     let typeParams = m.GetGenericArguments()
77                     let normalParams = m.GetParameters().Select(x => x.ParameterType)
78                     where typeParams.SequenceEqual(genericParameterTypes)
79                     && normalParams.SequenceEqual(argumentTypes)
80                     select m;
81             var method = methods.Single();
82             return method;
83         }
84     }
85 }

```

```

75     }
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static Type GetBaseType(this Type type) => type.BaseType;
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static Assembly GetAssembly(this Type type) => type.Assembly;
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static bool IsSubclassOf(this Type type, Type superClass) =>
85         type.IsSubclassOf(superClass);
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static bool IsValueType(this Type type) => type.IsValueType;
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public static bool IsGeneric(this Type type) => type.IsGenericType;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
95         type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    public static Type GetUnsignedVersionOrNull(this Type signedType) =>
102        _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
103
104    [MethodImpl(MethodImplOptions.AggressiveInlining)]
105    public static Type GetSignedVersionOrNull(this Type unsignedType) =>
106        _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);
107
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);
110
111    [MethodImpl(MethodImplOptions.AggressiveInlining)]
112    public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);
113
114    [MethodImpl(MethodImplOptions.AggressiveInlining)]
115    public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
116
117    [MethodImpl(MethodImplOptions.AggressiveInlining)]
118    public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
119
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    public static Type GetDelegateReturnType(this Type delegateType) =>
122        delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
123
124    [MethodImpl(MethodImplOptions.AggressiveInlining)]
125    public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
126        delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
127
128    [MethodImpl(MethodImplOptions.AggressiveInlining)]
129    public static void GetDelegateCharacteristics(this Type delegateType, out Type
130        return Type, out Type[] parameterTypes)
131    {
132        var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
133        return Type = invoke.ReturnType;
134        parameterTypes = invoke.GetParameterTypes();
135    }
136
137 }
138
139 }

```

1.13 ./csharp/Platform.Reflection/Types.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8  #pragma warning disable CA1819 // Properties should not return arrays
9
10 namespace Platform.Reflection
11 {
12     public abstract class Types
13     {
14         public static ReadOnlyCollection<Type> Collection { get; } = new
15             ReadOnlyCollection<Type>(System.Array.Empty<Type>());
16     }
17 }

```

```

15     public static Type[] Array => Collection.ToArray();
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected ReadOnlyCollection<Type> ToReadOnlyCollection()
19     {
20         var types = GetType().GetGenericArguments();
21         var result = new List<Type>();
22         AppendTypes(result, types);
23         return new ReadOnlyCollection<Type>(result);
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     private static void AppendTypes(List<Type> container, IList<Type> types)
28     {
29         for (var i = 0; i < types.Count; i++)
30         {
31             var element = types[i];
32             if (element != typeof(Types))
33             {
34                 if (element.IsSubclassOf(typeof(Types)))
35                 {
36                     AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>(
37                         nameof(Types.Collection)));
38                 }
39                 else
40                 {
41                     container.Add(element);
42                 }
43             }
44         }
45     }
46 }

```

1.14 ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4, T5, T6, T7>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }

```

1.15 ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5, T6> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
13             ↪ T4, T5, T6>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }

```

1.16 ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4, T5> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪ T4, T5>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.17 ./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3, T4> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3,
            ↪ T4>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.18 ./csharp/Platform.Reflection/Types[T1, T2, T3].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2, T3> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2,
            ↪ T3>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.19 ./csharp/Platform.Reflection/Types[T1, T2].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection
9  {
10     public class Types<T1, T2> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1,
            ↪ T2>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.20 ./csharp/Platform.Reflection/Types[T].cs

```

1  using System;
2  using System.Collections.ObjectModel;
3  using Platform.Collections.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  #pragma warning disable CA1819 // Properties should not return arrays
7
8  namespace Platform.Reflection

```

```

9  {
10     public class Types<T> : Types
11     {
12         public new static ReadOnlyCollection<Type> Collection { get; } = new
            ↳ Types<T>().ToReadOnlyCollection();
13         public new static Type[] Array => Collection.ToArray();
14         private Types() { }
15     }
16 }

```

1.21 ./csharp/Platform.Reflection.Tests/CodeGenerationTests.cs

```

1  using System;
2  using Xunit;
3
4  namespace Platform.Reflection.Tests
5  {
6      public class CodeGenerationTests
7      {
8          [Fact]
9          public void EmptyActionCompilationTest()
10         {
11             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
12             {
13                 generator.Return();
14             });
15             compiledAction();
16         }
17
18         [Fact]
19         public void FailedActionCompilationTest()
20         {
21             var compiledAction = DelegateHelpers.Compile<Action>(generator =>
22             {
23                 throw new NotImplementedException();
24             });
25             Assert.Throws<NotSupportedException>(compiledAction);
26         }
27
28         [Fact]
29         public void ConstantLoadingTest()
30         {
31             CheckConstantLoading<byte>(8);
32             CheckConstantLoading<uint>(8);
33             CheckConstantLoading<ushort>(8);
34             CheckConstantLoading<ulong>(8);
35         }
36
37         private void CheckConstantLoading<T>(T value)
38         {
39             var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
40             {
41                 generator.LoadConstant(value);
42                 generator.Return();
43             });
44             Assert.Equal(value, compiledFunction());
45         }
46
47         [Fact]
48         public void UnsignedIntegersConversionWithSignExtensionTest()
49         {
50             object[] withSignExtension = new object[]
51             {
52                 CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
53                 CompileUncheckedConverter<byte, short>(extendSign: true)(128),
54                 CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
55                 CompileUncheckedConverter<byte, int>(extendSign: true)(128),
56                 CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
57                 CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
58                 CompileUncheckedConverter<byte, long>(extendSign: true)(128),
59                 CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
60                 CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
61                 CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
62             };
63             object[] withoutSignExtension = new object[]
64             {
65                 CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
66                 CompileUncheckedConverter<byte, short>(extendSign: false)(128),
67                 CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),

```



```

68     CompileUncheckedConverter<byte, int>(extendSign: false)(128),
69     CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
70     CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
71     CompileUncheckedConverter<byte, long>(extendSign: false)(128),
72     CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
73     CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
74     CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
75 };
76 var i = 0;
77 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
78 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
79 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
80 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
81 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
82 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
83 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
84 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
85 Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
86 Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
87 }
88
89 [Fact]
90 public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
91 {
92     object[] withSignExtension = new object[]
93     {
94         CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
95         CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
96         CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
97         CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
98         CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
99         CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
100        CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
101        CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
102        CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
103        CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
104    };
105    object[] withoutSignExtension = new object[]
106    {
107        CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
108        CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
109        CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
110        CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
111        CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
112        CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
113        CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
114        CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
115        CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
116        CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
117    };
118    var i = 0;
119    Assert.Equal((byte)255, (byte)withSignExtension[i]);
120    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
121    Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
122    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
123    Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
124    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
125    Assert.Equal(4294967295, withSignExtension[i]);
126    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
127    Assert.Equal(4294967295, withSignExtension[i]);
128    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
129    Assert.Equal(4294967295, withSignExtension[i]);
130    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
131    Assert.Equal(18446744073709551615, withSignExtension[i]);
132    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
133    Assert.Equal(18446744073709551615, withSignExtension[i]);
134    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
135    Assert.Equal(18446744073709551615, withSignExtension[i]);
136    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
137    Assert.Equal(18446744073709551615, withSignExtension[i]);
138    Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
139 }
140
141 [Fact]
142 public void SignedIntegersConversionOfTwoWithSignExtensionTest()
143 {
144     object[] withSignExtension = new object[]
145     {

```

```

146     CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
147     CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
148     CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
149     CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
150     CompileUncheckedConverter<short, uint>(extendSign: true)(2),
151     CompileUncheckedConverter<int, uint>(extendSign: true)(2),
152     CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
153     CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
154     CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
155     CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
156 };
157 object[] withoutSignExtension = new object[]
158 {
159     CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
160     CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
161     CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
162     CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
163     CompileUncheckedConverter<short, uint>(extendSign: false)(2),
164     CompileUncheckedConverter<int, uint>(extendSign: false)(2),
165     CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
166     CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
167     CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
168     CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
169 };
170 for (var i = 0; i < withSignExtension.Length; i++)
171 {
172     Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
173     Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
174 }
175 }
176
177 private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
178     ↪ TTarget>(bool extendSign)
179 {
180     return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
181     {
182         generator.LoadArgument(0);
183         generator.UncheckedConvert<TSource, TTarget>(extendSign);
184         generator.Return();
185     });
186 }
187 }

```

1.22 ./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs

```

1  using System;
2  using System.Reflection;
3  using Xunit;
4  using Platform.Collections;
5  using Platform.Collections.Lists;
6
7  namespace Platform.Reflection.Tests
8  {
9      public static class GetILBytesMethodTests
10     {
11         [Fact]
12         public static void ILBytesForDelegateAreAvailableTest()
13         {
14             var function = new Func<object, int>(argument => 0);
15             var bytes = function.GetMethodInfo().GetILBytes();
16             Assert.False(bytes.IsNullOrEmpty());
17         }
18
19         [Fact]
20         public static void ILBytesForDifferentDelegatesAreTheSameTest()
21         {
22             var firstFunction = new Func<object, int>(argument => 0);
23             var secondFunction = new Func<object, int>(argument => 0);
24             Assert.False(firstFunction == secondFunction);
25             var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
26             Assert.False(firstFunctionBytes.IsNullOrEmpty());
27             var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
28             Assert.False(secondFunctionBytes.IsNullOrEmpty());
29             Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
30         }
31     }
32 }

```

1.23 ./csharp/Platform.Reflection.Tests/NumericTypeTests.cs

```
1  using Xunit;
2
3  namespace Platform.Reflection.Tests
4  {
5      public class NumericTypeTests
6      {
7          [Fact]
8          public void UInt64IsNumericTest()
9          {
10             Assert.True(NumericType<ulong>.IsNumeric);
11          }
12     }
13 }
```

Index

- ./csharp/Platform.Reflection.Tests/CodeGenerationTests.cs, 24
- ./csharp/Platform.Reflection.Tests/GetILBytesMethodTests.cs, 26
- ./csharp/Platform.Reflection.Tests/NumericTypeTests.cs, 26
- ./csharp/Platform.Reflection/AssemblyExtensions.cs, 1
- ./csharp/Platform.Reflection/DelegateHelpers.cs, 1
- ./csharp/Platform.Reflection/DynamicExtensions.cs, 2
- ./csharp/Platform.Reflection/EnsureExtensions.cs, 2
- ./csharp/Platform.Reflection/FieldInfoExtensions.cs, 5
- ./csharp/Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./csharp/Platform.Reflection/MethodInfoExtensions.cs, 17
- ./csharp/Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 17
- ./csharp/Platform.Reflection/NumericType.cs, 17
- ./csharp/Platform.Reflection/PropertyInfoExtensions.cs, 19
- ./csharp/Platform.Reflection/TypeBuilderExtensions.cs, 19
- ./csharp/Platform.Reflection/TypeExtensions.cs, 19
- ./csharp/Platform.Reflection/Types.cs, 21
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 22
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 22
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 22
- ./csharp/Platform.Reflection/Types[T1, T2, T3, T4].cs, 23
- ./csharp/Platform.Reflection/Types[T1, T2, T3].cs, 23
- ./csharp/Platform.Reflection/Types[T1, T2].cs, 23
- ./csharp/Platform.Reflection/Types[T].cs, 23