

LinksPlatform's Platform.Reflection Class Library

1.1 ./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 ./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             var @delegate = default(TDelegate);
41             try
42             {
43                 @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
44                     ↳ CompileDynamicMethod<TDelegate>(emitCode);
45             }
46             catch (Exception exception)
47             {
48                 exception.Ignore();
49             }
50             return @delegate;
51         }
52     }
53 }
```

```

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 ./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 ./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 ./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 ./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))
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))
103     {
104         generator.Emit(OpCodes.Conv_U2);
105     }
106     else if (targetType == typeof(int))
107     {
108         generator.Emit(OpCodes.Conv_I4);
109     }
110 }

```

```

109     else if (targetType == typeof(uint))
110     {
111         generator.Emit(OpCodes.Conv_U4);
112     }
113     else if (targetType == typeof(long) || targetType == typeof(ulong))
114     {
115         if (NumericType<TSource>.IsSigned || extendSign)
116         {
117             generator.Emit(OpCodes.Conv_I8);
118         }
119         else
120         {
121             generator.Emit(OpCodes.Conv_U8);
122         }
123     }
124 }
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public static void CheckedConvert<TSource, TTarget>(this ILGenerator generator)
128 {
129     var sourceType = typeof(TSource);
130     var targetType = typeof(TTarget);
131     if (sourceType == targetType)
132     {
133         return;
134     }
135     if (targetType == typeof(short))
136     {
137         if (NumericType<TSource>.IsSigned)
138         {
139             generator.Emit(OpCodes.Conv_Ovf_I2);
140         }
141         else
142         {
143             generator.Emit(OpCodes.Conv_Ovf_I2_Un);
144         }
145     }
146     else if (targetType == typeof(ushort))
147     {
148         if (NumericType<TSource>.IsSigned)
149         {
150             generator.Emit(OpCodes.Conv_Ovf_U2);
151         }
152         else
153         {
154             generator.Emit(OpCodes.Conv_Ovf_U2_Un);
155         }
156     }
157     else if (targetType == typeof(sbyte))
158     {
159         if (NumericType<TSource>.IsSigned)
160         {
161             generator.Emit(OpCodes.Conv_Ovf_I1);
162         }
163         else
164         {
165             generator.Emit(OpCodes.Conv_Ovf_I1_Un);
166         }
167     }
168     else if (targetType == typeof(byte))
169     {
170         if (NumericType<TSource>.IsSigned)
171         {
172             generator.Emit(OpCodes.Conv_Ovf_U1);
173         }
174         else
175         {
176             generator.Emit(OpCodes.Conv_Ovf_U1_Un);
177         }
178     }
179     else if (targetType == typeof(int))
180     {
181         if (NumericType<TSource>.IsSigned)
182         {
183             generator.Emit(OpCodes.Conv_Ovf_I4);
184         }
185         else
186         {

```

```

187         generator.Emit(OpCodes.Conv_Ovf_I4_Un);
188     }
189 }
190 else if (targetType == typeof(uint))
191 {
192     if (NumericType<TSource>.IsSigned)
193     {
194         generator.Emit(OpCodes.Conv_Ovf_U4);
195     }
196     else
197     {
198         generator.Emit(OpCodes.Conv_Ovf_U4_Un);
199     }
200 }
201 else if (targetType == typeof(long))
202 {
203     if (NumericType<TSource>.IsSigned)
204     {
205         generator.Emit(OpCodes.Conv_Ovf_I8);
206     }
207     else
208     {
209         generator.Emit(OpCodes.Conv_Ovf_I8_Un);
210     }
211 }
212 else if (targetType == typeof(ulong))
213 {
214     if (NumericType<TSource>.IsSigned)
215     {
216         generator.Emit(OpCodes.Conv_Ovf_U8);
217     }
218     else
219     {
220         generator.Emit(OpCodes.Conv_Ovf_U8_Un);
221     }
222 }
223 else if (targetType == typeof(float))
224 {
225     if (NumericType<TSource>.IsSigned)
226     {
227         generator.Emit(OpCodes.Conv_R4);
228     }
229     else
230     {
231         generator.Emit(OpCodes.Conv_R_Un);
232     }
233 }
234 else if (targetType == typeof(double))
235 {
236     generator.Emit(OpCodes.Conv_R8);
237 }
238 else if (targetType == typeof(bool))
239 {
240     generator.ConvertToBoolean<TSource>();
241 }
242 else
243 {
244     throw new NotSupportedException();
245 }
246 }
247
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static void LoadConstant(this ILGenerator generator, bool value) =>
250     ↪ generator.LoadConstant(value ? 1 : 0);
251
252 [MethodImpl(MethodImplOptions.AggressiveInlining)]
253 public static void LoadConstant(this ILGenerator generator, float value) =>
254     ↪ generator.Emit(OpCodes.Ldc_R4, value);
255
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static void LoadConstant(this ILGenerator generator, double value) =>
258     ↪ generator.Emit(OpCodes.Ldc_R8, value);
259
260 [MethodImpl(MethodImplOptions.AggressiveInlining)]
261 public static void LoadConstant(this ILGenerator generator, ulong value) =>
262     ↪ generator.Emit(OpCodes.Ldc_I8, unchecked((long)value));
263
264 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

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

```

```

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

```

```

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

```

```

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

```

```

563 {
564     if (isSigned)
565     {
566         generator.CompareGreaterThan();
567     }
568     else
569     {
570         generator.UnsignedCompareGreaterThan();
571     }
572 }
573
574 [MethodImpl(MethodImplOptions.AggressiveInlining)]
575 public static void CompareLessThan(this ILGenerator generator) =>
576     ↪ generator.Emit(OpCodes.Clt);
577
578 [MethodImpl(MethodImplOptions.AggressiveInlining)]
579 public static void UnsignedCompareLessThan(this ILGenerator generator) =>
580     ↪ generator.Emit(OpCodes.Clt_Un);
581
582 [MethodImpl(MethodImplOptions.AggressiveInlining)]
583 public static void CompareLessThan(this ILGenerator generator, bool isSigned)
584 {
585     if (isSigned)
586     {
587         generator.CompareLessThan();
588     }
589     else
590     {
591         generator.UnsignedCompareLessThan();
592     }
593 }
594
595 [MethodImpl(MethodImplOptions.AggressiveInlining)]
596 public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
597     ↪ generator.Emit(OpCodes.Bge, label);
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
601     ↪ label) => generator.Emit(OpCodes.Bge_Un, label);
602
603 [MethodImpl(MethodImplOptions.AggressiveInlining)]
604 public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
605     ↪ Label label)
606 {
607     if (isSigned)
608     {
609         generator.BranchIfGreaterOrEqual(label);
610     }
611     else
612     {
613         generator.UnsignedBranchIfGreaterOrEqual(label);
614     }
615 }
616
617 [MethodImpl(MethodImplOptions.AggressiveInlining)]
618 public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
619     ↪ generator.Emit(OpCodes.Ble, label);
620
621 [MethodImpl(MethodImplOptions.AggressiveInlining)]
622 public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
623     ↪ => generator.Emit(OpCodes.Ble_Un, label);
624
625 [MethodImpl(MethodImplOptions.AggressiveInlining)]
626 public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
627     ↪ label)
628 {
629     if (isSigned)
630     {
631         generator.BranchIfLessOrEqual(label);
632     }
633     else
634     {
635         generator.UnsignedBranchIfLessOrEqual(label);
636     }
637 }
638
639 [MethodImpl(MethodImplOptions.AggressiveInlining)]
640 public static void Box<TBox>(this ILGenerator generator) => generator.Box(typeof(TBox));

```

```

633 [MethodImpl(MethodImplOptions.AggressiveInlining)]
634 public static void Box(this ILGenerator generator, Type boxedType) =>
635     ↪ generator.Emit(OpCodes.Box, boxedType);
636
637 [MethodImpl(MethodImplOptions.AggressiveInlining)]
638 public static void Call(this ILGenerator generator, MethodInfo method) =>
639     ↪ generator.Emit(OpCodes.Call, method);
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
643
644 [MethodImpl(MethodImplOptions.AggressiveInlining)]
645 public static void Unbox<TUnbox>(this ILGenerator generator) =>
646     ↪ generator.Unbox(typeof(TUnbox));
647
648 [MethodImpl(MethodImplOptions.AggressiveInlining)]
649 public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
650     ↪ generator.Emit(OpCodes.Unbox, typeToUnbox);
651
652 [MethodImpl(MethodImplOptions.AggressiveInlining)]
653 public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
654     ↪ generator.UnboxValue(typeof(TUnbox));
655
656 [MethodImpl(MethodImplOptions.AggressiveInlining)]
657 public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
658     ↪ generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
659
660 [MethodImpl(MethodImplOptions.AggressiveInlining)]
661 public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
662     ↪ generator.DeclareLocal(typeof(T));
663
664 [MethodImpl(MethodImplOptions.AggressiveInlining)]
665 public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
666     ↪ generator.Emit(OpCodes.Ldloc, local);
667
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
670     ↪ generator.Emit(OpCodes.Stloc, local);
671
672 [MethodImpl(MethodImplOptions.AggressiveInlining)]
673 public static void NewObject(this ILGenerator generator, Type type, params Type[]
674     ↪ parameterTypes)
675 {
676     var allConstructors = type.GetConstructors(BindingFlags.Public |
677     ↪ BindingFlags.NonPublic | BindingFlags.Instance
678     ↪ | BindingFlags.CreateInstance
679     ↪ );
680     var constructor = allConstructors.Where(c => c.GetParameters().Length ==
681     ↪ parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
682     ↪ parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
683     if (constructor == null)
684     {
685         throw new InvalidOperationException("Type " + type + " must have a constructor
686         ↪ that matches parameters [" + string.Join(", ",
687         ↪ parameterTypes.AsEnumerable()) + "]");
688     }
689     generator.NewObject(constructor);
690 }
691
692 [MethodImpl(MethodImplOptions.AggressiveInlining)]
693 public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
694     ↪ generator.Emit(OpCodes.Newobj, constructor);
695
696 [MethodImpl(MethodImplOptions.AggressiveInlining)]
697 public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
698     ↪ byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
699
700 [MethodImpl(MethodImplOptions.AggressiveInlining)]
701 public static void LoadIndirect(this ILGenerator generator, Type type, bool isVolatile =
702     ↪ false, byte? unaligned = null)
703 {
704     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
705     {
706         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
707     }
708 }

```

```

692     }
693     if (isVolatile)
694     {
695         generator.Emit(OpCodes.Volatile);
696     }
697     if (unaligned.HasValue)
698     {
699         generator.Emit(OpCodes.Unaligned, unaligned.Value);
700     }
701     if (type.IsPointer)
702     {
703         generator.Emit(OpCodes.Ldind_I);
704     }
705     else if (!type.IsValueType)
706     {
707         generator.Emit(OpCodes.Ldind_Ref);
708     }
709     else if (type == typeof(sbyte))
710     {
711         generator.Emit(OpCodes.Ldind_I1);
712     }
713     else if (type == typeof(bool))
714     {
715         generator.Emit(OpCodes.Ldind_I1);
716     }
717     else if (type == typeof(byte))
718     {
719         generator.Emit(OpCodes.Ldind_U1);
720     }
721     else if (type == typeof(short))
722     {
723         generator.Emit(OpCodes.Ldind_I2);
724     }
725     else if (type == typeof(ushort))
726     {
727         generator.Emit(OpCodes.Ldind_U2);
728     }
729     else if (type == typeof(char))
730     {
731         generator.Emit(OpCodes.Ldind_U2);
732     }
733     else if (type == typeof(int))
734     {
735         generator.Emit(OpCodes.Ldind_I4);
736     }
737     else if (type == typeof(uint))
738     {
739         generator.Emit(OpCodes.Ldind_U4);
740     }
741     else if (type == typeof(long) || type == typeof(ulong))
742     {
743         generator.Emit(OpCodes.Ldind_I8);
744     }
745     else if (type == typeof(float))
746     {
747         generator.Emit(OpCodes.Ldind_R4);
748     }
749     else if (type == typeof(double))
750     {
751         generator.Emit(OpCodes.Ldind_R8);
752     }
753     else
754     {
755         throw new InvalidOperationException("LoadIndirect cannot be used with " + type +
756             ↪ ", LoadObject may be more appropriate");
757     }
758 }
759
760 [MethodImpl(MethodImplOptions.AggressiveInlining)]
761 public static void StoreIndirect<T>(this ILGenerator generator, bool isVolatile = false,
762     ↪ byte? unaligned = null) => generator.StoreIndirect(typeof(T), isVolatile, unaligned);
763
764 [MethodImpl(MethodImplOptions.AggressiveInlining)]
765 public static void StoreIndirect(this ILGenerator generator, Type type, bool isVolatile
766     ↪ = false, byte? unaligned = null)
767 {
768     if (unaligned.HasValue && unaligned != 1 && unaligned != 2 && unaligned != 4)
769     {

```

```

767         throw new ArgumentException("unaligned must be null, 1, 2, or 4");
768     }
769     if (isVolatile)
770     {
771         generator.Emit(OpCodes.Volatile);
772     }
773     if (unaligned.HasValue)
774     {
775         generator.Emit(OpCodes.Unaligned, unaligned.Value);
776     }
777     if (type.IsPointer)
778     {
779         generator.Emit(OpCodes.Stind_I);
780     }
781     else if (!type.IsValueType)
782     {
783         generator.Emit(OpCodes.Stind_Ref);
784     }
785     else if (type == typeof(sbyte) || type == typeof(byte))
786     {
787         generator.Emit(OpCodes.Stind_I1);
788     }
789     else if (type == typeof(short) || type == typeof(ushort))
790     {
791         generator.Emit(OpCodes.Stind_I2);
792     }
793     else if (type == typeof(int) || type == typeof(uint))
794     {
795         generator.Emit(OpCodes.Stind_I4);
796     }
797     else if (type == typeof(long) || type == typeof(ulong))
798     {
799         generator.Emit(OpCodes.Stind_I8);
800     }
801     else if (type == typeof(float))
802     {
803         generator.Emit(OpCodes.Stind_R4);
804     }
805     else if (type == typeof(double))
806     {
807         generator.Emit(OpCodes.Stind_R8);
808     }
809     else
810     {
811         throw new InvalidOperationException("StoreIndirect cannot be used with " + type
            ↪ + ", StoreObject may be more appropriate");
812     }
813 }
814 }
815 }

```

1.7 ./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) =>
            ↪ methodInfo.GetMethodBody().GetILAsByteArray();
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
            ↪ methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
17     }
18 }

```

1.8 ./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 ./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         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;

```

```

56         SignedVersion = signedVersion;
57         UnsignedVersion = unsignedVersion;
58     }
59     catch (Exception exception)
60     {
61         exception.Ignore();
62     }
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
67 {
68     if (type == typeof(bool))
69     {
70         minValue = (T)(object>false;
71         maxValue = (T)(object>true;
72     }
73     else
74     {
75         minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
76         maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
77     }
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
↪ signedVersion, out Type unsignedVersion)
82 {
83     if (isSigned)
84     {
85         signedVersion = type;
86         unsignedVersion = type.GetUnsignedVersionOrNull();
87     }
88     else
89     {
90         signedVersion = type.GetSignedVersionOrNull();
91         unsignedVersion = type;
92     }
93 }
94 }
95 }

```

1.10 ./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) =>
↪         (T)fieldInfo.GetValue(null);
12     }
13 }

```

1.11 ./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 =
↪         MethodAttributes.Public | MethodAttributes.Static;
13         public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes =
↪         MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final |
↪         MethodAttributes.HideBySig;
14         public static readonly MethodImplAttributes DefaultMethodImplAttributes =
↪         MethodImplAttributes.IL | MethodImplAttributes.Managed |
↪         MethodImplAttributes.AggressiveInlining;
15     }

```

```

16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName,
    ↳ MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes,
    ↳ Action<ILGenerator> emitCode)
18 {
19     typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[]
    ↳ parameterTypes);
20     EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType,
    ↳ parameterTypes, emitCode);
21 }
22
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes
    ↳ methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[]
    ↳ parameterTypes, Action<ILGenerator> emitCode)
25 {
26     MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType,
    ↳ parameterTypes);
27     method.SetImplementationFlags(methodImplAttributes);
28     var generator = method.GetILGenerator();
29     emitCode(generator);
30 }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName,
    ↳ Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
    ↳ DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string
    ↳ methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName,
    ↳ DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
37 }
38 }

```

1.12 ./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 |
    ↳ BindingFlags.NonPublic | BindingFlags.Static;
15         static public readonly string DefaultDelegateMethodName = "Invoke";
16
17         static private readonly HashSet<Type> _canBeNumericTypes;
18         static private readonly HashSet<Type> _isNumericTypes;
19         static private readonly HashSet<Type> _isSignedTypes;
20         static private readonly HashSet<Type> _isFloatPointTypes;
21         static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
22         static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         static TypeExtensions()
26         {
27             _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char),
    ↳ typeof(DateTime), typeof(TimeSpan) };
28             _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint),
    ↳ typeof(ulong) };
29             _canBeNumericTypes.UnionWith(_isNumericTypes);
30             _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int),
    ↳ typeof(long) };
31             _canBeNumericTypes.UnionWith(_isSignedTypes);
32             _isNumericTypes.UnionWith(_isSignedTypes);
33             _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double),
    ↳ typeof(float) };
34             _canBeNumericTypes.UnionWith(_isFloatPointTypes);
35             _isNumericTypes.UnionWith(_isFloatPointTypes);
36             _isSignedTypes.UnionWith(_isFloatPointTypes);
37             _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>

```

```

38     {
39         { typeof(sbyte), typeof(byte) },
40         { typeof(short), typeof(ushort) },
41         { typeof(int), typeof(uint) },
42         { typeof(long), typeof(ulong) },
43     };
44     signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
45     {
46         { typeof(byte), typeof(sbyte)},
47         { typeof(ushort), typeof(short) },
48         { typeof(uint), typeof(int) },
49         { typeof(ulong), typeof(long) },
50     };
51 }
52
53 [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public static T GetStaticFieldValue<T>(this Type type, string name) =>
58     ↳ type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static T GetStaticPropertyValue<T>(this Type type, string name) =>
62     ↳ type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static MethodInfo GetGenericMethod(this Type type, string name, Type[]
66     ↳ genericParameterTypes, Type[] argumentTypes)
67 {
68     var methods = from m in type.GetMethods()
69                   where m.Name == name
70                       && m.IsGenericMethodDefinition
71                       let typeParams = m.GetGenericArguments()
72                       let normalParams = m.GetParameters().Select(x => x.ParameterType)
73                       where typeParams.SequenceEqual(genericParameterTypes)
74                       && normalParams.SequenceEqual(argumentTypes)
75                       select m;
76     var method = methods.Single();
77     return method;
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public static Type GetBaseType(this Type type) => type.BaseType;
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public static Assembly GetAssembly(this Type type) => type.Assembly;
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public static bool IsSubclassOf(this Type type, Type superClass) =>
88     ↳ type.IsSubclassOf(superClass);
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool IsValueType(this Type type) => type.IsValueType;
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public static bool IsGeneric(this Type type) => type.IsGenericType;
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
98     ↳ type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 public static Type GetUnsignedVersionOrNull(this Type signedType) =>
105     ↳ _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 public static Type GetSignedVersionOrNull(this Type unsignedType) =>
109     ↳ _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

```

```

110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);
112
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);
115
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     public static Type GetDelegateReturnType(this Type delegateType) =>
118         ↳ delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;
119
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
122         ↳ delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();
123
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static void GetDelegateCharacteristics(this Type delegateType, out Type
126         ↳ returnType, out Type[] parameterTypes)
127     {
128         var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
129         returnType = invoke.ReturnType;
130         parameterTypes = invoke.GetParameterTypes();
131     }
132 }
133 }

```

1.13 ./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         public static Type[] Array => Collection.ToArray();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected ReadOnlyCollection<Type> ToReadOnlyCollection()
20         {
21             var types = GetType().GetGenericArguments();
22             var result = new List<Type>();
23             AppendTypes(result, types);
24             return new ReadOnlyCollection<Type>(result);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         private static void AppendTypes(List<Type> container, IList<Type> types)
29         {
30             for (var i = 0; i < types.Count; i++)
31             {
32                 var element = types[i];
33                 if (element != typeof(Types))
34                 {
35                     if (element.IsSubclassOf(typeof(Types)))
36                     {
37                         AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<
38                             ↳ <Type>>(nameof(Types<object>.Collection)));
39                     }
40                     else
41                     {
42                         container.Add(element);
43                     }
44                 }
45             }
46         }
47     }
48 }

```

1.14 ./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     }
17 }

```

1.15 ./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     }
17 }

```

1.16 ./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,
13             ↪ T4, T5>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

1.17 ./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,
13             ↪ T4>().ToReadOnlyCollection();
14         public new static Type[] Array => Collection.ToArray();
15         private Types() { }
16     }
17 }

```

1.18 ./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 ./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 ./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 ./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.Equal(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     });
185 }

```

```

183         generator.Return();
184     });
185 }
186 }
187 }

```

1.22 ./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 ./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

- ./Platform.Reflection.Tests/CodeGenerationTests.cs, 23
- ./Platform.Reflection.Tests/GetILBytesMethodTests.cs, 26
- ./Platform.Reflection.Tests/NumericTypeTests.cs, 26
- ./Platform.Reflection/AssemblyExtensions.cs, 1
- ./Platform.Reflection/DelegateHelpers.cs, 1
- ./Platform.Reflection/DynamicExtensions.cs, 2
- ./Platform.Reflection/EnsureExtensions.cs, 2
- ./Platform.Reflection/FieldInfoExtensions.cs, 5
- ./Platform.Reflection/ILGeneratorExtensions.cs, 5
- ./Platform.Reflection/MethodInfoExtensions.cs, 16
- ./Platform.Reflection/NotSupportedExceptionDelegateFactory.cs, 16
- ./Platform.Reflection/NumericType.cs, 17
- ./Platform.Reflection/PropertyInfoExtensions.cs, 18
- ./Platform.Reflection/TypeBuilderExtensions.cs, 18
- ./Platform.Reflection/TypeExtensions.cs, 19
- ./Platform.Reflection/Types.cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6, T7].cs, 21
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5, T6].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4, T5].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3, T4].cs, 22
- ./Platform.Reflection/Types[T1, T2, T3].cs, 22
- ./Platform.Reflection/Types[T1, T2].cs, 23
- ./Platform.Reflection/Types[T].cs, 23