LinksPlatform's Platform.Reflection Class Library

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

```
using System;
using System.Collections.Concurrent;
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class AssemblyExtensions
    {
        private static readonly ConcurrentDictionary<Assembly, Type[]> _loadableTypesCache = new
        ↪  ConcurrentDictionary<Assembly, Type[]>();

        /// <remarks>
        /// Source: http://haacked.com/archive/2012/07/23/get-all-types-in-an-assembly.aspx/
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetLoadableTypes(this Assembly assembly)
        {
            Ensure.Always.ArgumentNotNull(assembly, nameof(assembly));
            try
            {
                return assembly.GetTypes();
            }
            catch (ReflectionTypeLoadException e)
            {
                return e.Types.ToArray(t => t != null);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetCachedLoadableTypes(this Assembly assembly) =>
        ↪  _loadableTypesCache.GetOrAdd(assembly, GetLoadableTypes);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class DelegateHelpers
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
            where TDelegate : Delegate
        {
            var @delegate = default(TDelegate);
            try
            {
                @delegate = typeMemberMethod ? CompileTypeMemberMethod<TDelegate>(emitCode) :
                ↪  CompileDynamicMethod<TDelegate>(emitCode);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
            return @delegate;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileOrDefault<TDelegate>(Action<ILGenerator> emitCode) where
        ↪  TDelegate : Delegate => CompileOrDefault<TDelegate>(emitCode, false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode, bool
        ↪  typeMemberMethod)
```

```
            where TDelegate : Delegate
        {
            var @delegate = CompileOrDefault<TDelegate>(emitCode, typeMemberMethod);
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                @delegate = new NotSupportedExceptionDelegateFactory<TDelegate>().Create();
            }
            return @delegate;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate Compile<TDelegate>(Action<ILGenerator> emitCode) where TDelegate
        ↪  : Delegate => Compile<TDelegate>(emitCode, false);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileDynamicMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            var delegateType = typeof(TDelegate);
            delegateType.GetDelegateCharacteristics(out Type returnType, out Type[]
            ↪  parameterTypes);
            var dynamicMethod = new DynamicMethod(GetNewName(), returnType, parameterTypes);
            emitCode(dynamicMethod.GetILGenerator());
            return (TDelegate)(object)dynamicMethod.CreateDelegate(delegateType);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TDelegate CompileTypeMemberMethod<TDelegate>(Action<ILGenerator> emitCode)
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName());
            var methodName = GetNewName();
            type.EmitStaticMethod<TDelegate>(methodName, emitCode);
            var typeInfo = type.CreateTypeInfo();
            return (TDelegate)(object)typeInfo.GetMethod(methodName).CreateDelegate(typeof(TDele
            ↪  gate));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetNewName() => Guid.NewGuid().ToString("N");
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class DynamicExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool HasProperty(this object @object, string propertyName)
        {
            var type = @object.GetType();
            if (type is IDictionary<string, object> dictionary)
            {
                return dictionary.ContainsKey(propertyName);
            }
            return type.GetProperty(propertyName) != null;
        }
    }
}
```

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

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```
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,
            ↪  Func<string> messageBuilder)
18          {
19              if (!NumericType<T>.IsNumeric || NumericType<T>.IsSigned ||
                ↪  NumericType<T>.IsFloatPoint)
20              {
21                  throw new NotSupportedException(messageBuilder());
22              }
23          }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
            ↪  message)
27          {
28              string messageBuilder() => message;
29              IsUnsignedInteger<T>(root, messageBuilder);
30          }
31
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public static void IsUnsignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsUnsignedInteger<T>(root, (string)null);
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
37          {
38              if (!NumericType<T>.IsNumeric || !NumericType<T>.IsSigned ||
                ↪  NumericType<T>.IsFloatPoint)
39              {
40                  throw new NotSupportedException(messageBuilder());
41              }
42          }
43
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root, string
            ↪  message)
46          {
47              string messageBuilder() => message;
48              IsSignedInteger<T>(root, messageBuilder);
49          }
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          public static void IsSignedInteger<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsSignedInteger<T>(root, (string)null);
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
56          {
57              if (!NumericType<T>.IsSigned)
58              {
59                  throw new NotSupportedException(messageBuilder());
60              }
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root, string message)
65          {
66              string messageBuilder() => message;
67              IsSigned<T>(root, messageBuilder);
68          }
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public static void IsSigned<T>(this EnsureAlwaysExtensionRoot root) => IsSigned<T>(root,
            ↪  (string)null);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
75          {
76              if (!NumericType<T>.IsNumeric)
```

```
77              {
78                  throw new NotSupportedException(messageBuilder());
79              }
80          }
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
84          {
85              string messageBuilder() => message;
86              IsNumeric<T>(root, messageBuilder);
87          }
88
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          public static void IsNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  IsNumeric<T>(root, (string)null);
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, Func<string>
            ↪  messageBuilder)
94          {
95              if (!NumericType<T>.CanBeNumeric)
96              {
97                  throw new NotSupportedException(messageBuilder());
98              }
99          }
100
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root, string message)
103         {
104             string messageBuilder() => message;
105             CanBeNumeric<T>(root, messageBuilder);
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public static void CanBeNumeric<T>(this EnsureAlwaysExtensionRoot root) =>
            ↪  CanBeNumeric<T>(root, (string)null);
110
111         #endregion
112
113         #region OnDebug
114
115         [Conditional("DEBUG")]
116         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root,
            ↪  Func<string> messageBuilder) => Ensure.Always.IsUnsignedInteger<T>(messageBuilder);
117
118         [Conditional("DEBUG")]
119         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsUnsignedInteger<T>(message);
120
121         [Conditional("DEBUG")]
122         public static void IsUnsignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsUnsignedInteger<T>();
123
124         [Conditional("DEBUG")]
125         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSignedInteger<T>(messageBuilder);
126
127         [Conditional("DEBUG")]
128         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root, string
            ↪  message) => Ensure.Always.IsSignedInteger<T>(message);
129
130         [Conditional("DEBUG")]
131         public static void IsSignedInteger<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSignedInteger<T>();
132
133         [Conditional("DEBUG")]
134         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, Func<string>
            ↪  messageBuilder) => Ensure.Always.IsSigned<T>(messageBuilder);
135
136         [Conditional("DEBUG")]
137         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root, string message) =>
            ↪  Ensure.Always.IsSigned<T>(message);
138
139         [Conditional("DEBUG")]
140         public static void IsSigned<T>(this EnsureOnDebugExtensionRoot root) =>
            ↪  Ensure.Always.IsSigned<T>();
141
142         [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                 {
```

```
                    generator.Emit(OpCodes.Conv_I1);
                }
                if (sourceType == typeof(ushort) || sourceType == typeof(char))
                {
                    generator.Emit(OpCodes.Conv_I2);
                }
            }
            if (NumericType<TSource>.BitsSize > NumericType<TTarget>.BitsSize)
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign: false);
            }
            else
            {
                generator.ConvertToInteger<TSource>(targetType, extendSign);
            }
            if (targetType == typeof(float))
            {
                if (NumericType<TSource>.IsSigned)
                {
                    generator.Emit(OpCodes.Conv_R4);
                }
                else
                {
                    generator.Emit(OpCodes.Conv_R_Un);
                }
            }
            else if (targetType == typeof(double))
            {
                generator.Emit(OpCodes.Conv_R8);
            }
            else if (targetType == typeof(bool))
            {
                generator.ConvertToBoolean<TSource>();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToBoolean<TSource>(this ILGenerator generator)
        {
            generator.LoadConstant<TSource>(default);
            var sourceType = typeof(TSource);
            if (sourceType == typeof(float) || sourceType == typeof(double))
            {
                generator.Emit(OpCodes.Ceq);
                // Inversion of the first Ceq instruction
                generator.LoadConstant<int>(0);
                generator.Emit(OpCodes.Ceq);
            }
            else
            {
                generator.Emit(OpCodes.Cgt_Un);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ConvertToInteger<TSource>(this ILGenerator generator, Type
        ↪  targetType, bool extendSign)
        {
            if (targetType == typeof(sbyte))
            {
                generator.Emit(OpCodes.Conv_I1);
            }
            else if (targetType == typeof(byte))
            {
                generator.Emit(OpCodes.Conv_U1);
            }
            else if (targetType == typeof(short))
            {
                generator.Emit(OpCodes.Conv_I2);
            }
            else if (targetType == typeof(ushort) || targetType == typeof(char))
            {
                var sourceType = typeof(TSource);
                if (sourceType != typeof(ushort) && sourceType != typeof(char))
                {
                    generator.Emit(OpCodes.Conv_U2);
                }
            }
```

```
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) =>
            ↪ generator.LoadConstant(value ? 1 : 0);
257 
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         public static void LoadConstant(this ILGenerator generator, float value) =>
            ↪ generator.Emit(OpCodes.Ldc_R4, value);
260 
261         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```
                    return;
                case 5:
                    generator.Emit(OpCodes.Ldc_I4_5);
                    return;
                case 6:
                    generator.Emit(OpCodes.Ldc_I4_6);
                    return;
                case 7:
                    generator.Emit(OpCodes.Ldc_I4_7);
                    return;
                case 8:
                    generator.Emit(OpCodes.Ldc_I4_8);
                    return;
                default:
                    if (value >= sbyte.MinValue && value <= sbyte.MaxValue)
                    {
                        generator.Emit(OpCodes.Ldc_I4_S, unchecked((byte)value));
                    }
                    else
                    {
                        generator.Emit(OpCodes.Ldc_I4, value);
                    }
                    return;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, short value) =>
        ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, ushort value) =>
        ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, sbyte value) =>
        ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstant(this ILGenerator generator, byte value) =>
        ↪  generator.LoadConstant((int)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne<TConstant>(this ILGenerator generator) =>
        ↪  LoadConstantOne(generator, typeof(TConstant));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void LoadConstantOne(this ILGenerator generator, Type constantType)
        {
            if (constantType == typeof(float))
            {
                generator.LoadConstant(1F);
            }
            else if (constantType == typeof(double))
            {
                generator.LoadConstant(1D);
            }
            else if (constantType == typeof(long))
            {
                generator.LoadConstant(1L);
            }
            else if (constantType == typeof(ulong))
            {
                generator.LoadConstant(1UL);
            }
            else if (constantType == typeof(int))
            {
                generator.LoadConstant(1);
            }
            else if (constantType == typeof(uint))
            {
                generator.LoadConstant(1U);
            }
            else if (constantType == typeof(short))
            {
                generator.LoadConstant((short)1);
            }
            else if (constantType == typeof(ushort))
            {
```

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

```
564
565         [MethodImpl(MethodImplOptions.AggressiveInlining)]
566         public static void UnsignedCompareGreaterThan(this ILGenerator generator) =>
            ↪  generator.Emit(OpCodes.Cgt_Un);
567
568         [MethodImpl(MethodImplOptions.AggressiveInlining)]
569         public static void CompareGreaterThan(this ILGenerator generator, bool isSigned)
570         {
571             if (isSigned)
572             {
573                 generator.CompareGreaterThan();
574             }
575             else
576             {
577                 generator.UnsignedCompareGreaterThan();
578             }
579         }
580
581         [MethodImpl(MethodImplOptions.AggressiveInlining)]
582         public static void CompareLessThan(this ILGenerator generator) =>
            ↪  generator.Emit(OpCodes.Clt);
583
584         [MethodImpl(MethodImplOptions.AggressiveInlining)]
585         public static void UnsignedCompareLessThan(this ILGenerator generator) =>
            ↪  generator.Emit(OpCodes.Clt_Un);
586
587         [MethodImpl(MethodImplOptions.AggressiveInlining)]
588         public static void CompareLessThan(this ILGenerator generator, bool isSigned)
589         {
590             if (isSigned)
591             {
592                 generator.CompareLessThan();
593             }
594             else
595             {
596                 generator.UnsignedCompareLessThan();
597             }
598         }
599
600         [MethodImpl(MethodImplOptions.AggressiveInlining)]
601         public static void BranchIfGreaterOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Bge, label);
602
603         [MethodImpl(MethodImplOptions.AggressiveInlining)]
604         public static void UnsignedBranchIfGreaterOrEqual(this ILGenerator generator, Label
            ↪  label) => generator.Emit(OpCodes.Bge_Un, label);
605
606         [MethodImpl(MethodImplOptions.AggressiveInlining)]
607         public static void BranchIfGreaterOrEqual(this ILGenerator generator, bool isSigned,
            ↪  Label label)
608         {
609             if (isSigned)
610             {
611                 generator.BranchIfGreaterOrEqual(label);
612             }
613             else
614             {
615                 generator.UnsignedBranchIfGreaterOrEqual(label);
616             }
617         }
618
619         [MethodImpl(MethodImplOptions.AggressiveInlining)]
620         public static void BranchIfLessOrEqual(this ILGenerator generator, Label label) =>
            ↪  generator.Emit(OpCodes.Ble, label);
621
622         [MethodImpl(MethodImplOptions.AggressiveInlining)]
623         public static void UnsignedBranchIfLessOrEqual(this ILGenerator generator, Label label)
            ↪  => generator.Emit(OpCodes.Ble_Un, label);
624
625         [MethodImpl(MethodImplOptions.AggressiveInlining)]
626         public static void BranchIfLessOrEqual(this ILGenerator generator, bool isSigned, Label
            ↪  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) =>
        ↪  generator.Emit(OpCodes.Box, boxedType);
643 
644         [MethodImpl(MethodImplOptions.AggressiveInlining)]
645         public static void Call(this ILGenerator generator, MethodInfo method) =>
        ↪  generator.Emit(OpCodes.Call, method);
646 
647         [MethodImpl(MethodImplOptions.AggressiveInlining)]
648         public static void Return(this ILGenerator generator) => generator.Emit(OpCodes.Ret);
649 
650         [MethodImpl(MethodImplOptions.AggressiveInlining)]
651         public static void Unbox<TUnbox>(this ILGenerator generator) =>
        ↪  generator.Unbox(typeof(TUnbox));
652 
653         [MethodImpl(MethodImplOptions.AggressiveInlining)]
654         public static void Unbox(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox, typeToUnbox);
655 
656         [MethodImpl(MethodImplOptions.AggressiveInlining)]
657         public static void UnboxValue<TUnbox>(this ILGenerator generator) =>
        ↪  generator.UnboxValue(typeof(TUnbox));
658 
659         [MethodImpl(MethodImplOptions.AggressiveInlining)]
660         public static void UnboxValue(this ILGenerator generator, Type typeToUnbox) =>
        ↪  generator.Emit(OpCodes.Unbox_Any, typeToUnbox);
661 
662         [MethodImpl(MethodImplOptions.AggressiveInlining)]
663         public static LocalBuilder DeclareLocal<T>(this ILGenerator generator) =>
        ↪  generator.DeclareLocal(typeof(T));
664 
665         [MethodImpl(MethodImplOptions.AggressiveInlining)]
666         public static void LoadLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪  generator.Emit(OpCodes.Ldloc, local);
667 
668         [MethodImpl(MethodImplOptions.AggressiveInlining)]
669         public static void StoreLocal(this ILGenerator generator, LocalBuilder local) =>
        ↪  generator.Emit(OpCodes.Stloc, local);
670 
671         [MethodImpl(MethodImplOptions.AggressiveInlining)]
672         public static void NewObject(this ILGenerator generator, Type type, params Type[]
        ↪  parameterTypes)
673         {
674             var allConstructors = type.GetConstructors(BindingFlags.Public |
            ↪  BindingFlags.NonPublic | BindingFlags.Instance
675 #if !NETSTANDARD
676                 | BindingFlags.CreateInstance
677 #endif
678                 );
679             var constructor = allConstructors.Where(c => c.GetParameters().Length ==
            ↪  parameterTypes.Length && c.GetParameters().Select((p, i) => p.ParameterType ==
            ↪  parameterTypes[i]).Aggregate(true, (a, b) => a && b)).SingleOrDefault();
680             if (constructor == null)
681             {
682                 throw new InvalidOperationException("Type " + type + " must have a constructor
                ↪  that matches parameters [" + string.Join(", ",
                ↪  parameterTypes.AsEnumerable()) + "]");
683             }
684             generator.NewObject(constructor);
685         }
686 
687         [MethodImpl(MethodImplOptions.AggressiveInlining)]
688         public static void NewObject(this ILGenerator generator, ConstructorInfo constructor) =>
        ↪  generator.Emit(OpCodes.Newobj, constructor);
689 
690         [MethodImpl(MethodImplOptions.AggressiveInlining)]
691         public static void LoadIndirect<T>(this ILGenerator generator, bool isVolatile = false,
        ↪  byte? unaligned = null) => generator.LoadIndirect(typeof(T), isVolatile, unaligned);
692 
693         [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)
```

```
            {
                generator.Emit(OpCodes.Div);
            }
            else
            {
                generator.Emit(OpCodes.Div_Un);
            }
        }
    }
}
```

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

```
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class MethodInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] GetILBytes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetMethodBody().GetILAsByteArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetParameterTypes(this MethodInfo methodInfo) =>
        ↪  methodInfo.GetParameters().Select(p => p.ParameterType).ToArray();
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public class NotSupportedExceptionDelegateFactory<TDelegate> : IFactory<TDelegate>
        where TDelegate : Delegate
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TDelegate Create()
        {
            var @delegate = DelegateHelpers.CompileOrDefault<TDelegate>(generator =>
            {
                generator.Throw<NotSupportedException>();
            });
            if (EqualityComparer<TDelegate>.Default.Equals(@delegate, default))
            {
                throw new InvalidOperationException("Unable to compile stub delegate.");
            }
            return @delegate;
        }
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Platform.Exceptions;

// ReSharper disable AssignmentInConditionalExpression
// ReSharper disable BuiltInTypeReferenceStyle
// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class NumericType<T>
    {
        public static readonly Type Type;
        public static readonly Type UnderlyingType;
```

```
        public static readonly Type SignedVersion;
        public static readonly Type UnsignedVersion;
        public static readonly bool IsFloatPoint;
        public static readonly bool IsNumeric;
        public static readonly bool IsSigned;
        public static readonly bool CanBeNumeric;
        public static readonly bool IsNullable;
        public static readonly int BytesSize;
        public static readonly int BitsSize;
        public static readonly T MinValue;
        public static readonly T MaxValue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static NumericType()
        {
            try
            {
                var type = typeof(T);
                var isNullable = type.IsNullable();
                var underlyingType = isNullable ? Nullable.GetUnderlyingType(type) : type;
                var canBeNumeric = underlyingType.CanBeNumeric();
                var isNumeric = underlyingType.IsNumeric();
                var isSigned = underlyingType.IsSigned();
                var isFloatPoint = underlyingType.IsFloatPoint();
                var bytesSize = Marshal.SizeOf(underlyingType);
                var bitsSize = bytesSize * 8;
                GetMinAndMaxValues(underlyingType, out T minValue, out T maxValue);
                GetSignedAndUnsignedVersions(underlyingType, isSigned, out Type signedVersion,
                ↪  out Type unsignedVersion);
                Type = type;
                IsNullable = isNullable;
                UnderlyingType = underlyingType;
                CanBeNumeric = canBeNumeric;
                IsNumeric = isNumeric;
                IsSigned = isSigned;
                IsFloatPoint = isFloatPoint;
                BytesSize = bytesSize;
                BitsSize = bitsSize;
                MinValue = minValue;
                MaxValue = maxValue;
                SignedVersion = signedVersion;
                UnsignedVersion = unsignedVersion;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetMinAndMaxValues(Type type, out T minValue, out T maxValue)
        {
            if (type == typeof(bool))
            {
                minValue = (T)(object)false;
                maxValue = (T)(object)true;
            }
            else
            {
                minValue = type.GetStaticFieldValue<T>(nameof(int.MinValue));
                maxValue = type.GetStaticFieldValue<T>(nameof(int.MaxValue));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetSignedAndUnsignedVersions(Type type, bool isSigned, out Type
        ↪  signedVersion, out Type unsignedVersion)
        {
            if (isSigned)
            {
                signedVersion = type;
                unsignedVersion = type.GetUnsignedVersionOrNull();
            }
            else
            {
                signedVersion = type.GetSignedVersionOrNull();
                unsignedVersion = type;
            }
        }
    }
```

```
}
```

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

```
using System.Reflection;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class PropertyInfoExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticValue<T>(this PropertyInfo fieldInfo) => (T)fieldInfo.GetValue(null);
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;

namespace Platform.Reflection
{
    public static class TypeBuilderExtensions
    {
        public static readonly MethodAttributes DefaultStaticMethodAttributes = MethodAttributes.Public | MethodAttributes.Static;
        public static readonly MethodAttributes DefaultFinalVirtualMethodAttributes = MethodAttributes.Public | MethodAttributes.Virtual | MethodAttributes.Final | MethodAttributes.HideBySig;
        public static readonly MethodImplAttributes DefaultMethodImplAttributes = MethodImplAttributes.IL | MethodImplAttributes.Managed | MethodImplAttributes.AggressiveInlining;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod<TDelegate>(this TypeBuilder type, string methodName, MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes, Action<ILGenerator> emitCode)
        {
            typeof(TDelegate).GetDelegateCharacteristics(out Type returnType, out Type[] parameterTypes);
            EmitMethod(type, methodName, methodAttributes, methodImplAttributes, returnType, parameterTypes, emitCode);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitMethod(this TypeBuilder type, string methodName, MethodAttributes methodAttributes, MethodImplAttributes methodImplAttributes, Type returnType, Type[] parameterTypes, Action<ILGenerator> emitCode)
        {
            MethodBuilder method = type.DefineMethod(methodName, methodAttributes, returnType, parameterTypes);
            method.SetImplementationFlags(methodImplAttributes);
            var generator = method.GetILGenerator();
            emitCode(generator);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitStaticMethod<TDelegate>(this TypeBuilder type, string methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName, DefaultStaticMethodAttributes, DefaultMethodImplAttributes, emitCode);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EmitFinalVirtualMethod<TDelegate>(this TypeBuilder type, string methodName, Action<ILGenerator> emitCode) => type.EmitMethod<TDelegate>(methodName, DefaultFinalVirtualMethodAttributes, DefaultMethodImplAttributes, emitCode);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Reflection;
using System.Runtime.CompilerServices;
using Platform.Collections;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Reflection
{
    public static class TypeExtensions
    {
        static public readonly BindingFlags StaticMemberBindingFlags = BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Static;
        static public readonly string DefaultDelegateMethodName = "Invoke";

        static private readonly HashSet<Type> _canBeNumericTypes;
        static private readonly HashSet<Type> _isNumericTypes;
        static private readonly HashSet<Type> _isSignedTypes;
        static private readonly HashSet<Type> _isFloatPointTypes;
        static private readonly Dictionary<Type, Type> _unsignedVersionsOfSignedTypes;
        static private readonly Dictionary<Type, Type> _signedVersionsOfUnsignedTypes;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static TypeExtensions()
        {
            _canBeNumericTypes = new HashSet<Type> { typeof(bool), typeof(char), typeof(DateTime), typeof(TimeSpan) };
            _isNumericTypes = new HashSet<Type> { typeof(byte), typeof(ushort), typeof(uint), typeof(ulong) };
            _canBeNumericTypes.UnionWith(_isNumericTypes);
            _isSignedTypes = new HashSet<Type> { typeof(sbyte), typeof(short), typeof(int), typeof(long) };
            _canBeNumericTypes.UnionWith(_isSignedTypes);
            _isNumericTypes.UnionWith(_isSignedTypes);
            _isFloatPointTypes = new HashSet<Type> { typeof(decimal), typeof(double), typeof(float) };
            _canBeNumericTypes.UnionWith(_isFloatPointTypes);
            _isNumericTypes.UnionWith(_isFloatPointTypes);
            _isSignedTypes.UnionWith(_isFloatPointTypes);
            _unsignedVersionsOfSignedTypes = new Dictionary<Type, Type>
            {
                { typeof(sbyte), typeof(byte) },
                { typeof(short), typeof(ushort) },
                { typeof(int), typeof(uint) },
                { typeof(long), typeof(ulong) },
            };
            _signedVersionsOfUnsignedTypes = new Dictionary<Type, Type>
            {
                { typeof(byte), typeof(sbyte)},
                { typeof(ushort), typeof(short) },
                { typeof(uint), typeof(int) },
                { typeof(ulong), typeof(long) },
            };
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FieldInfo GetFirstField(this Type type) => type.GetFields()[0];

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticFieldValue<T>(this Type type, string name) =>
            type.GetField(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetStaticPropertyValue<T>(this Type type, string name) =>
            type.GetProperty(name, StaticMemberBindingFlags).GetStaticValue<T>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static MethodInfo GetGenericMethod(this Type type, string name, Type[] genericParameterTypes, Type[] argumentTypes)
        {
            var methods = from m in type.GetMethods()
                          where m.Name == name
                             && m.IsGenericMethodDefinition
                          let typeParams = m.GetGenericArguments()
                          let normalParams = m.GetParameters().Select(x => x.ParameterType)
                          where typeParams.SequenceEqual(genericParameterTypes)
                             && normalParams.SequenceEqual(argumentTypes)
                          select m;
            var method = methods.Single();
            return method;
```

```
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetBaseType(this Type type) => type.BaseType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Assembly GetAssembly(this Type type) => type.Assembly;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSubclassOf(this Type type, Type superClass) =>
        ↪  type.IsSubclassOf(superClass);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsValueType(this Type type) => type.IsValueType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type) => type.IsGenericType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsGeneric(this Type type, Type genericTypeDefinition) =>
        ↪  type.IsGeneric() && type.GetGenericTypeDefinition() == genericTypeDefinition;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullable(this Type type) => type.IsGeneric(typeof(Nullable<>));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetUnsignedVersionOrNull(this Type signedType) =>
        ↪  _unsignedVersionsOfSignedTypes.GetOrDefault(signedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetSignedVersionOrNull(this Type unsignedType) =>
        ↪  _signedVersionsOfUnsignedTypes.GetOrDefault(unsignedType);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool CanBeNumeric(this Type type) => _canBeNumericTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNumeric(this Type type) => _isNumericTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsSigned(this Type type) => _isSignedTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFloatPoint(this Type type) => _isFloatPointTypes.Contains(type);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type GetDelegateReturnType(this Type delegateType) =>
        ↪  delegateType.GetMethod(DefaultDelegateMethodName).ReturnType;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Type[] GetDelegateParameterTypes(this Type delegateType) =>
        ↪  delegateType.GetMethod(DefaultDelegateMethodName).GetParameterTypes();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetDelegateCharacteristics(this Type delegateType, out Type
        ↪  returnType, out Type[] parameterTypes)
        {
            var invoke = delegateType.GetMethod(DefaultDelegateMethodName);
            returnType = invoke.ReturnType;
            parameterTypes = invoke.GetParameterTypes();
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Runtime.CompilerServices;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public abstract class Types
    {
        public static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  ReadOnlyCollection<Type>(System.Array.Empty<Type>());
```

```
        public static Type[] Array => Collection.ToArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected ReadOnlyCollection<Type> ToReadOnlyCollection()
        {
            var types = GetType().GetGenericArguments();
            var result = new List<Type>();
            AppendTypes(result, types);
            return new ReadOnlyCollection<Type>(result);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendTypes(List<Type> container, IList<Type> types)
        {
            for (var i = 0; i < types.Count; i++)
            {
                var element = types[i];
                if (element != typeof(Types))
                {
                    if (element.IsSubclassOf(typeof(Types)))
                    {
                        AppendTypes(container, element.GetStaticPropertyValue<ReadOnlyCollection<Type>>(nameof(Types<object>.Collection)));
                    }
                    else
                    {
                        container.Add(element);
                    }
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5, T6, T7> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5, T6, T7>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5, T6> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5, T6>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4, T5> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4, T5>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3, T4> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3, T4>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2, T3> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2, T3>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
{
    public class Types<T1, T2> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new Types<T1, T2>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using System.Collections.ObjectModel;
using Platform.Collections.Lists;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable CA1819 // Properties should not return arrays

namespace Platform.Reflection
```

```
{
    public class Types<T> : Types
    {
        public new static ReadOnlyCollection<Type> Collection { get; } = new
        ↪  Types<T>().ToReadOnlyCollection();
        public new static Type[] Array => Collection.ToArray();
        private Types() { }
    }
}
```

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

```
using System;
using Xunit;

namespace Platform.Reflection.Tests
{
    public class CodeGenerationTests
    {
        [Fact]
        public void EmptyActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                generator.Return();
            });
            compiledAction();
        }

        [Fact]
        public void FailedActionCompilationTest()
        {
            var compiledAction = DelegateHelpers.Compile<Action>(generator =>
            {
                throw new NotImplementedException();
            });
            Assert.Throws<NotSupportedException>(compiledAction);
        }

        [Fact]
        public void ConstantLoadingTest()
        {
            CheckConstantLoading<byte>(8);
            CheckConstantLoading<uint>(8);
            CheckConstantLoading<ushort>(8);
            CheckConstantLoading<ulong>(8);
        }

        private void CheckConstantLoading<T>(T value)
        {
            var compiledFunction = DelegateHelpers.Compile<Func<T>>(generator =>
            {
                generator.LoadConstant(value);
                generator.Return();
            });
            Assert.Equal(value, compiledFunction());
        }

        [Fact]
        public void UnsignedIntegersConversionWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: true)(128),
                CompileUncheckedConverter<byte, short>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: true)(32768),
                CompileUncheckedConverter<byte, int>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: true)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: true)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: true)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: true)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: true)(9223372036854775808)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<byte, sbyte>(extendSign: false)(128),
                CompileUncheckedConverter<byte, short>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, short>(extendSign: false)(32768),
```

```
                CompileUncheckedConverter<byte, int>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, int>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, int>(extendSign: false)(2147483648),
                CompileUncheckedConverter<byte, long>(extendSign: false)(128),
                CompileUncheckedConverter<ushort, long>(extendSign: false)(32768),
                CompileUncheckedConverter<uint, long>(extendSign: false)(2147483648),
                CompileUncheckedConverter<ulong, long>(extendSign: false)(9223372036854775808)
            };
            var i = 0;
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.NotEqual(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
        }

        [Fact]
        public void SignedIntegersConversionOfMinusOneWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ushort>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: true)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(-1)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(-1),
                CompileUncheckedConverter<short, ushort>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<short, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<int, uint>(extendSign: false)(-1),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<short, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<int, ulong>(extendSign: false)(-1),
                CompileUncheckedConverter<long, ulong>(extendSign: false)(-1)
            };
            var i = 0;
            Assert.Equal((byte)255, (byte)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal((ushort)65535, (ushort)withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(4294967295, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
            Assert.Equal(18446744073709551615, withSignExtension[i]);
            Assert.Equal(withSignExtension[i], withoutSignExtension[i++]);
        }

        [Fact]
        public void SignedIntegersConversionOfTwoWithSignExtensionTest()
        {
            object[] withSignExtension = new object[]
            {
```

```
                CompileUncheckedConverter<sbyte, byte>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: true)(2),
                CompileUncheckedConverter<short, ushort>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, uint>(extendSign: true)(2),
                CompileUncheckedConverter<short, uint>(extendSign: true)(2),
                CompileUncheckedConverter<int, uint>(extendSign: true)(2),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<short, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<int, ulong>(extendSign: true)(2),
                CompileUncheckedConverter<long, ulong>(extendSign: true)(2)
            };
            object[] withoutSignExtension = new object[]
            {
                CompileUncheckedConverter<sbyte, byte>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, ushort>(extendSign: false)(2),
                CompileUncheckedConverter<short, ushort>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, uint>(extendSign: false)(2),
                CompileUncheckedConverter<short, uint>(extendSign: false)(2),
                CompileUncheckedConverter<int, uint>(extendSign: false)(2),
                CompileUncheckedConverter<sbyte, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<short, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<int, ulong>(extendSign: false)(2),
                CompileUncheckedConverter<long, ulong>(extendSign: false)(2)
            };
            for (var i = 0; i < withSignExtension.Length; i++)
            {
                Assert.Equal(2UL, Convert.ToUInt64(withSignExtension[i]));
                Assert.Equal(withSignExtension[i], withoutSignExtension[i]);
            }
        }

        private static Converter<TSource, TTarget> CompileUncheckedConverter<TSource,
        ↪  TTarget>(bool extendSign)
        {
            return DelegateHelpers.Compile<Converter<TSource, TTarget>>(generator =>
            {
                generator.LoadArgument(0);
                generator.UncheckedConvert<TSource, TTarget>(extendSign);
                generator.Return();
            });
        }
    }
}
```

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

```
using System;
using System.Reflection;
using Xunit;
using Platform.Collections;
using Platform.Collections.Lists;

namespace Platform.Reflection.Tests
{
    public static class GetILBytesMethodTests
    {
        [Fact]
        public static void ILBytesForDelegateAreAvailableTest()
        {
            var function = new Func<object, int>(argument => 0);
            var bytes = function.GetMethodInfo().GetILBytes();
            Assert.False(bytes.IsNullOrEmpty());
        }

        [Fact]
        public static void ILBytesForDifferentDelegatesAreTheSameTest()
        {
            var firstFunction = new Func<object, int>(argument => 0);
            var secondFunction = new Func<object, int>(argument => 0);
            Assert.False(firstFunction == secondFunction);
            var firstFunctionBytes = firstFunction.GetMethodInfo().GetILBytes();
            Assert.False(firstFunctionBytes.IsNullOrEmpty());
            var secondFunctionBytes = secondFunction.GetMethodInfo().GetILBytes();
            Assert.False(secondFunctionBytes.IsNullOrEmpty());
            Assert.True(firstFunctionBytes.EqualTo(secondFunctionBytes));
        }
    }
}
```

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

```
using Xunit;

namespace Platform.Reflection.Tests
{
    public class NumericTypeTests
    {
        [Fact]
        public void UInt64IsNumericTest()
        {
            Assert.True(NumericType<ulong>.IsNumeric);
        }
    }
}
```

Index