### 1.1 ./csharp/Platform.Data.Doublets.Sequences/Converters/BalancedVariantConverter.cs

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the balanced variant converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksListToSequenceConverterBase{TLink}"/>
    public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BalancedVariantConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Converts the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length < 1)
            {
                return default;
            }
            if (length == 1)
            {
                return sequence[0];
            }
            // Make copy of next layer
            if (length > 2)
            {
                // TODO: Try to use stackalloc (which at the moment is not working with
                ↪  generics) but will be possible with Sigil
                var halvedSequence = new TLink[(length / 2) + (length % 2)];
                HalveSequence(halvedSequence, sequence, length);
                sequence = halvedSequence;
                length = halvedSequence.Length;
            }
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }

        /// <summary>
        /// <para>
```

```
76          /// Halves the sequence using the specified destination.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <param name="destination">
81          /// <para>The destination.</para>
82          /// <para></para>
83          /// </param>
84          /// <param name="source">
85          /// <para>The source.</para>
86          /// <para></para>
87          /// </param>
88          /// <param name="length">
89          /// <para>The length.</para>
90          /// <para></para>
91          /// </param>
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
94          {
95              var loopedLength = length - (length % 2);
96              for (var i = 0; i < loopedLength; i += 2)
97              {
98                  destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
99              }
100             if (length > loopedLength)
101             {
102                 destination[length / 2] = source[length - 1];
103             }
104         }
105     }
106 }
```

## 1.2 ./csharp/Platform.Data.Doublets.Sequences/Converters/CompressingConverter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections;
5   using Platform.Converters;
6   using Platform.Singletons;
7   using Platform.Numbers;
8   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets.Sequences.Converters
13  {
14      /// <remarks>
15      /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
        ↪  Links на этапе сжатия.
16      ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
        ↪  таком случае тип значения элемента массива может быть любым, как char так и ulong.
17      ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
        ↪  пар, а так же разом выполнить замену.
18      /// </remarks>
19      public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
20      {
21          /// <summary>
22          /// <para>
23          /// The instance.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          private static readonly LinksConstants<TLink> _constants =
            ↪  Default<LinksConstants<TLink>>.Instance;
28          /// <summary>
29          /// <para>
30          /// The default.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
35          /// <summary>
36          /// <para>
37          /// The default.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
```

```
42
43          /// <summary>
44          /// <para>
45          /// The zero.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          private static readonly TLink _zero = default;
50          /// <summary>
51          /// <para>
52          /// The zero.
53          /// </para>
54          /// <para></para>
55          /// </summary>
56          private static readonly TLink _one = Arithmetic.Increment(_zero);
57
58          /// <summary>
59          /// <para>
60          /// The base converter.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          private readonly IConverter<IList<TLink>, TLink> _baseConverter;
65          /// <summary>
66          /// <para>
67          /// The doublet frequencies cache.
68          /// </para>
69          /// <para></para>
70          /// </summary>
71          private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
72          /// <summary>
73          /// <para>
74          /// The min frequency to compress.
75          /// </para>
76          /// <para></para>
77          /// </summary>
78          private readonly TLink _minFrequencyToCompress;
79          /// <summary>
80          /// <para>
81          /// The do initial frequencies increment.
82          /// </para>
83          /// <para></para>
84          /// </summary>
85          private readonly bool _doInitialFrequenciesIncrement;
86          /// <summary>
87          /// <para>
88          /// The max doublet.
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          private Doublet<TLink> _maxDoublet;
93          /// <summary>
94          /// <para>
95          /// The max doublet data.
96          /// </para>
97          /// <para></para>
98          /// </summary>
99          private LinkFrequency<TLink> _maxDoubletData;
100
101         /// <summary>
102         /// <para>
103         /// The half doublet.
104         /// </para>
105         /// <para></para>
106         /// </summary>
107         private struct HalfDoublet
108         {
109             /// <summary>
110             /// <para>
111             /// The element.
112             /// </para>
113             /// <para></para>
114             /// </summary>
115             public TLink Element;
116             /// <summary>
117             /// <para>
118             /// The doublet data.
119             /// </para>
```

```
            /// <para></para>
            /// </summary>
            public LinkFrequency<TLink> DoubletData;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="HalfDoublet"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="element">
            /// <para>A element.</para>
            /// <para></para>
            /// </param>
            /// <param name="doubletData">
            /// <para>A doublet data.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            /// <summary>
            /// <para>
            /// Returns the string.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <returns>
            /// <para>The string</para>
            /// <para></para>
            /// </returns>
            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CompressingConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="doubletFrequenciesCache">
        /// <para>A doublet frequencies cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CompressingConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="doubletFrequenciesCache">
        /// <para>A doublet frequencies cache.</para>
        /// <para></para>
```

```
197         /// </param>
198         /// <param name="doInitialFrequenciesIncrement">
199         /// <para>A do initial frequencies increment.</para>
200         /// <para></para>
201         /// </param>
202         [MethodImpl(MethodImplOptions.AggressiveInlining)]
203         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
            ↪ doInitialFrequenciesIncrement)
204             : this(links, baseConverter, doubletFrequenciesCache, _one,
                ↪ doInitialFrequenciesIncrement) { }
205 
206         /// <summary>
207         /// <para>
208         /// Initializes a new <see cref="CompressingConverter"/> instance.
209         /// </para>
210         /// <para></para>
211         /// </summary>
212         /// <param name="links">
213         /// <para>A links.</para>
214         /// <para></para>
215         /// </param>
216         /// <param name="baseConverter">
217         /// <para>A base converter.</para>
218         /// <para></para>
219         /// </param>
220         /// <param name="doubletFrequenciesCache">
221         /// <para>A doublet frequencies cache.</para>
222         /// <para></para>
223         /// </param>
224         /// <param name="minFrequencyToCompress">
225         /// <para>A min frequency to compress.</para>
226         /// <para></para>
227         /// </param>
228         /// <param name="doInitialFrequenciesIncrement">
229         /// <para>A do initial frequencies increment.</para>
230         /// <para></para>
231         /// </param>
232         [MethodImpl(MethodImplOptions.AggressiveInlining)]
233         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
234             : base(links)
235         {
236             _baseConverter = baseConverter;
237             _doubletFrequenciesCache = doubletFrequenciesCache;
238             if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
239             {
240                 minFrequencyToCompress = _one;
241             }
242             _minFrequencyToCompress = minFrequencyToCompress;
243             _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
244             ResetMaxDoublet();
245         }
246 
247         /// <summary>
248         /// <para>
249         /// Converts the source.
250         /// </para>
251         /// <para></para>
252         /// </summary>
253         /// <param name="source">
254         /// <para>The source.</para>
255         /// <para></para>
256         /// </param>
257         /// <returns>
258         /// <para>The link</para>
259         /// <para></para>
260         /// </returns>
261         [MethodImpl(MethodImplOptions.AggressiveInlining)]
262         public override TLink Convert(IList<TLink> source) =>
            ↪ _baseConverter.Convert(Compress(source));
263 
264         /// <remarks>
265         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
266         /// Faster version (doublets' frequencies dictionary is not recreated).
267         /// </remarks>
268         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
269         private IList<TLink> Compress(IList<TLink> sequence)
270         {
271             if (sequence.IsNullOrEmpty())
272             {
273                 return null;
274             }
275             if (sequence.Count == 1)
276             {
277                 return sequence;
278             }
279             if (sequence.Count == 2)
280             {
281                 return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
282             }
283             // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
284             var copy = new HalfDoublet[sequence.Count];
285             Doublet<TLink> doublet = default;
286             for (var i = 1; i < sequence.Count; i++)
287             {
288                 doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
289                 LinkFrequency<TLink> data;
290                 if (_doInitialFrequenciesIncrement)
291                 {
292                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
293                 }
294                 else
295                 {
296                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
297                     if (data == null)
298                     {
299                         throw new NotSupportedException("If you ask not to increment
                            ↪ frequencies, it is expected that all frequencies for the sequence
                            ↪ are prepared.");
300                     }
301                 }
302                 copy[i - 1].Element = sequence[i - 1];
303                 copy[i - 1].DoubletData = data;
304                 UpdateMaxDoublet(ref doublet, data);
305             }
306             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
307             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
308             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
309             {
310                 var newLength = ReplaceDoublets(copy);
311                 sequence = new TLink[newLength];
312                 for (int i = 0; i < newLength; i++)
313                 {
314                     sequence[i] = copy[i].Element;
315                 }
316             }
317             return sequence;
318         }
319 
320         /// <remarks>
321         /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
322         /// </remarks>
323         [MethodImpl(MethodImplOptions.AggressiveInlining)]
324         private int ReplaceDoublets(HalfDoublet[] copy)
325         {
326             var oldLength = copy.Length;
327             var newLength = copy.Length;
328             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
329             {
330                 var maxDoubletSource = _maxDoublet.Source;
331                 var maxDoubletTarget = _maxDoublet.Target;
332                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
333                 {
334                     _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                        ↪ maxDoubletTarget);
335                 }
336                 var maxDoubletReplacementLink = _maxDoubletData.Link;
337                 oldLength--;
338                 var oldLengthMinusTwo = oldLength - 1;
339                 // Substitute all usages
340                 int w = 0, r = 0; // (r == read, w == write)
341                 for (; r < oldLength; r++)
342                 {
343                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                        ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
```

```
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪ maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                            ↪ xDoubletReplacementLink,
                            ↪ next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }

        /// <summary>
        /// <para>
        /// Resets the max doublet.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLink>();
            _maxDoubletData = new LinkFrequency<TLink>();
        }

        /// <summary>
        /// <para>
        /// Updates the max doublet using the specified copy.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="copy">
        /// <para>The copy.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
        {
            Doublet<TLink> doublet = default;
            for (var i = 1; i < length; i++)
            {
                doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
                UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
            }
        }

        /// <summary>
        /// <para>
        /// Updates the max doublet using the specified doublet.
```

```
418         /// </para>
419         /// <para></para>
420         /// </summary>
421         /// <param name="doublet">
422         /// <para>The doublet.</para>
423         /// <para></para>
424         /// </param>
425         /// <param name="data">
426         /// <para>The data.</para>
427         /// <para></para>
428         /// </param>
429         [MethodImpl(MethodImplOptions.AggressiveInlining)]
430         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
431         {
432             var frequency = data.Frequency;
433             var maxFrequency = _maxDoubletData.Frequency;
434             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
435             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
436                (_comparer.Compare(maxFrequency, frequency) < 0 ||
                    ↪  (_equalityComparer.Equals(maxFrequency, frequency) &&
                    ↪  _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                    ↪  Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                    ↪  better stability and better compression on sequent data and even on rundom
                    ↪  numbers data (but gives collisions anyway) */
437             {
438                 _maxDoublet = doublet;
439                 _maxDoubletData = data;
440             }
441         }
442     }
443 }
```

## 1.3 ./csharp/Platform.Data.Doublets.Sequences/Converters/LinksListToSequenceConverterBase.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Converters
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the links list to sequence converter base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksOperatorBase{TLink}"/>
16     /// <seealso cref="IConverter{IList{TLink}, TLink}"/>
17     public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
       ↪  IConverter<IList<TLink>, TLink>
18     {
19         /// <summary>
20         /// <para>
21         /// Initializes a new <see cref="LinksListToSequenceConverterBase"/> instance.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         /// <param name="links">
26         /// <para>A links.</para>
27         /// <para></para>
28         /// </param>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
31
32         /// <summary>
33         /// <para>
34         /// Converts the source.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         /// <param name="source">
39         /// <para>The source.</para>
40         /// <para></para>
41         /// </param>
```

```
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLink Convert(IList<TLink> source);
    }
}
```

### 1.4 ./csharp/Platform.Data.Doublets.Sequences/Converters/OptimalVariantConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Lists;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the optimal variant converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksListToSequenceConverterBase{TLink}"/>
    public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The sequence to its local element levels converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceToItsLocalElementLevelsConverter">
        /// <para>A sequence to its local element levels converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
        ↪  sequenceToItsLocalElementLevelsConverter) : base(links)
            => _sequenceToItsLocalElementLevelsConverter =
            ↪  sequenceToItsLocalElementLevelsConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkFrequenciesCache">
        /// <para>A link frequencies cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
        ↪  linkFrequenciesCache)
            : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
            ↪  ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
            ↪  }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLink> links)
            : this(links, new LinkFrequenciesCache<TLink>(links, new
            ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }

        /// <summary>
        /// <para>
        /// Converts the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Convert(IList<TLink> sequence)
        {
            var length = sequence.Count;
            if (length == 1)
            {
                return sequence[0];
            }
            if (length == 2)
            {
                return _links.GetOrCreate(sequence[0], sequence[1]);
            }
            sequence = sequence.ToArray();
            var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
            while (length > 2)
            {
                var levelRepeat = 1;
                var currentLevel = levels[0];
                var previousLevel = levels[0];
                var skipOnce = false;
                var w = 0;
                for (var i = 1; i < length; i++)
                {
                    if (_equalityComparer.Equals(currentLevel, levels[i]))
                    {
                        levelRepeat++;
                        skipOnce = false;
                        if (levelRepeat == 2)
                        {
                            sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
                            var newLevel = i >= length - 1 ?
                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel) :
```

```
                                i < 2 ?
                                GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }

        /// <summary>
        /// <para>
        /// Gets the greatest neigbour lower than current or current using the specified
        ↪  previous.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="previous">
        /// <para>The previous.</para>
        /// <para></para>
        /// </param>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <param name="next">
        /// <para>The next.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪  current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }

        /// <summary>
        /// <para>
```

```
        /// Gets the next lower than current or current using the specified current.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <param name="next">
        /// <para>The next.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪  _comparer.Compare(next, current) < 0 ? next : current;

        /// <summary>
        /// <para>
        /// Gets the previous lower than current or current using the specified previous.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="previous">
        /// <para>The previous.</para>
        /// <para></para>
        /// </param>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

## 1.5 ./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the sequence to its local element levels converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{IList{TLink}}"/>
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The link to its frequency to number conveter.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
33          private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
34
35          /// <summary>
36          /// <para>
37          /// Initializes a new <see cref="SequenceToItsLocalElementLevelsConverter"/> instance.
38          /// </para>
39          /// <para></para>
40          /// </summary>
41          /// <param name="links">
42          /// <para>A links.</para>
43          /// <para></para>
44          /// </param>
45          /// <param name="linkToItsFrequencyToNumberConveter">
46          /// <para>A link to its frequency to number conveter.</para>
47          /// <para></para>
48          /// </param>
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
            ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
            ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
51
52          /// <summary>
53          /// <para>
54          /// Converts the sequence.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          /// <param name="sequence">
59          /// <para>The sequence.</para>
60          /// <para></para>
61          /// </param>
62          /// <returns>
63          /// <para>The levels.</para>
64          /// <para></para>
65          /// </returns>
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public IList<TLink> Convert(IList<TLink> sequence)
68          {
69              var levels = new TLink[sequence.Count];
70              levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
71              for (var i = 1; i < sequence.Count - 1; i++)
72              {
73                  var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
74                  var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
75                  levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
76              }
77              levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
                ↪  sequence[sequence.Count - 1]);
78              return levels;
79          }
80
81          /// <summary>
82          /// <para>
83          /// Gets the frequency number using the specified source.
84          /// </para>
85          /// <para></para>
86          /// </summary>
87          /// <param name="source">
88          /// <para>The source.</para>
89          /// <para></para>
90          /// </param>
91          /// <param name="target">
92          /// <para>The target.</para>
93          /// <para></para>
94          /// </param>
95          /// <returns>
96          /// <para>The link</para>
97          /// <para></para>
98          /// </returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public TLink GetFrequencyNumber(TLink source, TLink target) =>
            ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
101     }
102 }
```

## 1.6 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
```

```

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

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    /// <summary>
    /// <para>
    /// Represents the default sequence element criterion matcher.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ICriterionMatcher{TLink}"/>
    public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
    ↪  ICriterionMatcher<TLink>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceElementCriterionMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
    }
}
```

## 1.7 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

```
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.Data.Doublets.Sequences.CriterionMatchers
{
    /// <summary>
    /// <para>
    /// Represents the marked sequence criterion matcher.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICriterionMatcher{TLink}"/>
    public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
```

```
30          /// <para></para>
31          /// </summary>
32          private readonly ILinks<TLink> _links;
33          /// <summary>
34          /// <para>
35          /// The sequence marker link.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          private readonly TLink _sequenceMarkerLink;
40
41          /// <summary>
42          /// <para>
43          /// Initializes a new <see cref="MarkedSequenceCriterionMatcher"/> instance.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          /// <param name="links">
48          /// <para>A links.</para>
49          /// <para></para>
50          /// </param>
51          /// <param name="sequenceMarkerLink">
52          /// <para>A sequence marker link.</para>
53          /// <para></para>
54          /// </param>
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
57          {
58              _links = links;
59              _sequenceMarkerLink = sequenceMarkerLink;
60          }
61
62          /// <summary>
63          /// <para>
64          /// Determines whether this instance is matched.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="sequenceCandidate">
69          /// <para>The sequence candidate.</para>
70          /// <para></para>
71          /// </param>
72          /// <returns>
73          /// <para>The bool</para>
74          /// <para></para>
75          /// </returns>
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          public bool IsMatched(TLink sequenceCandidate)
78              => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
79              || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
                ↪  sequenceCandidate), _links.Constants.Null);
80      }
81  }
```

## 1.8 ./csharp/Platform.Data.Doublets.Sequences/DefaultSequenceAppender.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4   using Platform.Data.Doublets.Sequences.HeightProviders;
5   using Platform.Data.Sequences;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences
10  {
11      /// <summary>
12      /// <para>
13      /// Represents the default sequence appender.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      /// <seealso cref="LinksOperatorBase{TLink}"/>
18      /// <seealso cref="ISequenceAppender{TLink}"/>
19      public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
        ↪  ISequenceAppender<TLink>
20      {
21          /// <summary>
22          /// <para>
23          /// The default.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The stack.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IStack<TLink> _stack;
        /// <summary>
        /// <para>
        /// The height provider.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ISequenceHeightProvider<TLink> _heightProvider;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceAppender"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="heightProvider">
        /// <para>A height provider.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↪  ISequenceHeightProvider<TLink> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        /// <summary>
        /// <para>
        /// Appends the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="appendant">
        /// <para>The appendant.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Append(TLink sequence, TLink appendant)
        {
            var cursor = sequence;
            var links = _links;
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = links.GetSource(cursor);
                var target = links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
```

```
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
            ↪  links.Constants.Null))
            {
                right = links.GetOrCreate(left, right);
                left = cursor;
            }
            return links.GetOrCreate(left, right);
        }
    }
}
```

## 1.9 ./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsCounter.cs

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the duplicate segments counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{int}"/>
    public class DuplicateSegmentsCounter<TLink> : ICounter<int>
    {
        /// <summary>
        /// <para>
        /// The duplicate fragments provider.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
        ↪  _duplicateFragmentsProvider;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DuplicateSegmentsCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="duplicateFragmentsProvider">
        /// <para>A duplicate fragments provider.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

## 1.10 ./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsProvider.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Singletons;
using Platform.Converters;
using Platform.Data.Doublets.Unicode;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the duplicate segments provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLink}"/>
    /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLink}, IList{TLink}}}}"/>
    public class DuplicateSegmentsProvider<TLink> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
    ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
        ↪  UncheckedConverter<TLink, long>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪  UncheckedConverter<TLink, ulong>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪  UncheckedConverter<ulong, TLink>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<TLink> _links;
        /// <summary>
        /// <para>
        /// The sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<TLink> _sequences;
        /// <summary>
        /// <para>
        /// The groups.
        /// </para>
        /// <para></para>
        /// </summary>
        private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
        /// <summary>
        /// <para>
        /// The visited.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        private BitString _visited;

        /// <summary>
        /// <para>
        /// Represents the item equility comparer.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="IEqualityComparer{KeyValuePair{IList{TLink}, IList{TLink}}}"/>
        private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
        ↪  IList<TLink>>>
        {
            /// <summary>
            /// <para>
            /// The list comparer.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly IListEqualityComparer<TLink> _listComparer;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            public ItemEquilityComparer() => _listComparer =
            ↪  Default<IListEqualityComparer<TLink>>.Instance;

            /// <summary>
            /// <para>
            /// Determines whether this instance equals.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="left">
            /// <para>The left.</para>
            /// <para></para>
            /// </param>
            /// <param name="right">
            /// <para>The right.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪  KeyValuePair<IList<TLink>, IList<TLink>> right) =>
            ↪  _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
            ↪  right.Value);

            /// <summary>
            /// <para>
            /// Gets the hash code using the specified pair.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="pair">
            /// <para>The pair.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The int</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
            ↪  (_listComparer.GetHashCode(pair.Key),
            ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
        }

        /// <summary>
        /// <para>
        /// Represents the item comparer.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="IComparer{KeyValuePair{IList{TLink}, IList{TLink}}}"/>
        private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
        {
            /// <summary>
            /// <para>
            /// The list comparer.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly IListComparer<TLink> _listComparer;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="ItemComparer"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;

            /// <summary>
            /// <para>
            /// Compares the left.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="left">
            /// <para>The left.</para>
            /// <para></para>
            /// </param>
            /// <param name="right">
            /// <para>The right.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The intermediate result.</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
            ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
            {
                var intermediateResult = _listComparer.Compare(left.Key, right.Key);
                if (intermediateResult == 0)
                {
                    intermediateResult = _listComparer.Compare(left.Value, right.Value);
                }
                return intermediateResult;
            }
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DuplicateSegmentsProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequences">
        /// <para>A sequences.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
            : base(minimumStringSegmentLength: 2)
        {
            _links = links;
            _sequences = sequences;
        }

        /// <summary>
        /// <para>
```

```
223         /// Gets this instance.
224         /// </para>
225         /// <para></para>
226         /// </summary>
227         /// <returns>
228         /// <para>The result list.</para>
229         /// <para></para>
230         /// </returns>
231         [MethodImpl(MethodImplOptions.AggressiveInlining)]
232         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
233         {
234             _groups = new HashSet<KeyValuePair<IList<TLink>,
                    ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
235             var links = _links;
236             var count = links.Count();
237             _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
238             links.Each(link =>
239             {
240                 var linkIndex = links.GetIndex(link);
241                 var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
242                 var constants = links.Constants;
243                 if (!_visited.Get(linkBitIndex))
244                 {
245                     var sequenceElements = new List<TLink>();
246                     var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
247                     _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                            ↪  LinkAddress<TLink>(linkIndex));
248                     if (sequenceElements.Count > 2)
249                     {
250                         WalkAll(sequenceElements);
251                     }
252                 }
253                 return constants.Continue;
254             });
255             var resultList = _groups.ToList();
256             var comparer = Default<ItemComparer>.Instance;
257             resultList.Sort(comparer);
258 #if DEBUG
259             foreach (var item in resultList)
260             {
261                 PrintDuplicates(item);
262             }
263 #endif
264             return resultList;
265         }
266 
267         /// <summary>
268         /// <para>
269         /// Creates the segment using the specified elements.
270         /// </para>
271         /// <para></para>
272         /// </summary>
273         /// <param name="elements">
274         /// <para>The elements.</para>
275         /// <para></para>
276         /// </param>
277         /// <param name="offset">
278         /// <para>The offset.</para>
279         /// <para></para>
280         /// </param>
281         /// <param name="length">
282         /// <para>The length.</para>
283         /// <para></para>
284         /// </param>
285         /// <returns>
286         /// <para>A segment of t link</para>
287         /// <para></para>
288         /// </returns>
289         [MethodImpl(MethodImplOptions.AggressiveInlining)]
290         protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
                ↪  length) => new Segment<TLink>(elements, offset, length);
291 
292         /// <summary>
293         /// <para>
294         /// Ons the dublicate found using the specified segment.
295         /// </para>
296         /// <para></para>
297         /// </summary>
```

```
298         /// <param name="segment">
299         /// <para>The segment.</para>
300         /// <para></para>
301         /// </param>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         protected override void OnDublicateFound(Segment<TLink> segment)
304         {
305             var duplicates = CollectDuplicatesForSegment(segment);
306             if (duplicates.Count > 1)
307             {
308                 _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
309             }
310         }
311 
312         /// <summary>
313         /// <para>
314         /// Collects the duplicates for segment using the specified segment.
315         /// </para>
316         /// <para></para>
317         /// </summary>
318         /// <param name="segment">
319         /// <para>The segment.</para>
320         /// <para></para>
321         /// </param>
322         /// <returns>
323         /// <para>The duplicates.</para>
324         /// <para></para>
325         /// </returns>
326         [MethodImpl(MethodImplOptions.AggressiveInlining)]
327         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
328         {
329             var duplicates = new List<TLink>();
330             var readAsElement = new HashSet<TLink>();
331             var restrictions = segment.ShiftRight();
332             var constants = _links.Constants;
333             restrictions[0] = constants.Any;
334             _sequences.Each(sequence =>
335             {
336                 var sequenceIndex = sequence[constants.IndexPart];
337                 duplicates.Add(sequenceIndex);
338                 readAsElement.Add(sequenceIndex);
339                 return constants.Continue;
340             }, restrictions);
341             if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
342             {
343                 return new List<TLink>();
344             }
345             foreach (var duplicate in duplicates)
346             {
347                 var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
348                 _visited.Set(duplicateBitIndex);
349             }
350             if (_sequences is Sequences sequencesExperiments)
351             {
352                 var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
                    ↪  ashSet<ulong>)(object)readAsElement,
                    ↪  (IList<ulong>)segment);
353                 foreach (var partiallyMatchedSequence in partiallyMatched)
354                 {
355                     var sequenceIndex =
                        ↪  _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
356                     duplicates.Add(sequenceIndex);
357                 }
358             }
359             duplicates.Sort();
360             return duplicates;
361         }
362 
363         /// <summary>
364         /// <para>
365         /// Prints the duplicates using the specified duplicates item.
366         /// </para>
367         /// <para></para>
368         /// </summary>
369         /// <param name="duplicatesItem">
370         /// <para>The duplicates item.</para>
371         /// <para></para>
```

```
372         /// </param>
373         [MethodImpl(MethodImplOptions.AggressiveInlining)]
374         private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
375         {
376             if (!(_links is ILinks<ulong> ulongLinks))
377             {
378                 return;
379             }
380             var duplicatesKey = duplicatesItem.Key;
381             var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
382             Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
383             var duplicatesList = duplicatesItem.Value;
384             for (int i = 0; i < duplicatesList.Count; i++)
385             {
386                 var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
387                 var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
                    ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
                    ↪ UnicodeMap.IsCharLink(link.Index) ?
                    ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
388                 Console.WriteLine(formatedSequenceStructure);
389                 var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
                    ↪ ulongLinks);
390                 Console.WriteLine(sequenceString);
391             }
392             Console.WriteLine();
393         }
394     }
395 }
```

## 1.11 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
       ↪ between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
17         /// <summary>
18         /// <para>
19         /// The default.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
24         /// <summary>
25         /// <para>
26         /// The default.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         /// <summary>
33         /// <para>
34         /// The zero.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         private static readonly TLink _zero = default;
39         /// <summary>
40         /// <para>
41         /// The zero.
42         /// </para>
43         /// <para></para>
44         /// </summary>
45         private static readonly TLink _one = Arithmetic.Increment(_zero);
46
47         /// <summary>
```

```
48          /// <para>
49          /// The doublets cache.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
54          /// <summary>
55          /// <para>
56          /// The frequency counter.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          private readonly ICounter<TLink, TLink> _frequencyCounter;
61
62          /// <summary>
63          /// <para>
64          /// Initializes a new <see cref="LinkFrequenciesCache"/> instance.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="links">
69          /// <para>A links.</para>
70          /// <para></para>
71          /// </param>
72          /// <param name="frequencyCounter">
73          /// <para>A frequency counter.</para>
74          /// <para></para>
75          /// </param>
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
78              : base(links)
79          {
80              _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                ↪  DoubletComparer<TLink>.Default);
81              _frequencyCounter = frequencyCounter;
82          }
83
84          /// <summary>
85          /// <para>
86          /// Gets the frequency using the specified source.
87          /// </para>
88          /// <para></para>
89          /// </summary>
90          /// <param name="source">
91          /// <para>The source.</para>
92          /// <para></para>
93          /// </param>
94          /// <param name="target">
95          /// <para>The target.</para>
96          /// <para></para>
97          /// </param>
98          /// <returns>
99          /// <para>A link frequency of t link</para>
100         /// <para></para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
104         {
105             var doublet = new Doublet<TLink>(source, target);
106             return GetFrequency(ref doublet);
107         }
108
109         /// <summary>
110         /// <para>
111         /// Gets the frequency using the specified doublet.
112         /// </para>
113         /// <para></para>
114         /// </summary>
115         /// <param name="doublet">
116         /// <para>The doublet.</para>
117         /// <para></para>
118         /// </param>
119         /// <returns>
120         /// <para>The data.</para>
121         /// <para></para>
122         /// </returns>
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
```

```
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
            return data;
        }

        /// <summary>
        /// <para>
        /// Increments the frequencies using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Increments the frequency using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A link frequency of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
        {
            var doublet = new Doublet<TLink>(source, target);
            return IncrementFrequency(ref doublet);
        }

        /// <summary>
        /// <para>
        /// Prints the frequencies using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequencies(IList<TLink> sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                PrintFrequency(sequence[i - 1], sequence[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Prints the frequency using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintFrequency(TLink source, TLink target)
        {
            var number = GetFrequency(source, target).Frequency;
            Console.WriteLine("({0},{1}) - {2}", source, target, number);
        }

        /// <summary>
        /// <para>
        /// Increments the frequency using the specified doublet.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="doublet">
        /// <para>The doublet.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The data.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
        {
            if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
            {
                data.IncrementFrequency();
            }
            else
            {
                var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
                data = new LinkFrequency<TLink>(_one, link);
                if (!_equalityComparer.Equals(link, default))
                {
                    data.Frequency = Arithmetic.Add(data.Frequency,
                    ↪  _frequencyCounter.Count(link));
                }
                _doubletsCache.Add(doublet, data);
            }
            return data;
        }

        /// <summary>
        /// <para>
        /// Validates the frequencies.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Frequencies validation failed.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                    ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
                     || ((_comparer.Compare(count, frequency) > 0) &&
                      ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
```

```
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new InvalidOperationException("Frequencies validation
                ↪  failed.");
                //    }
                //}
            }
        }
    }
}
```

## 1.12 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequency.cs

```
using System.Runtime.CompilerServices;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <summary>
    /// <para>
    /// Represents the link frequency.
    /// </para>
    /// <para></para>
    /// </summary>
    public class LinkFrequency<TLink>
    {
        /// <summary>
        /// <para>
        /// Gets or sets the frequency value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLink Frequency { get; set; }
        /// <summary>
        /// <para>
        /// Gets or sets the link value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLink Link { get; set; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkFrequency"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="frequency">
        /// <para>A frequency.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency(TLink frequency, TLink link)
        {
            Frequency = frequency;
            Link = link;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkFrequency"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency() { }

        /// <summary>
```

```
        /// <para>
        /// Increments the frequency.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);

        /// <summary>
        /// <para>
        /// Decrements the frequency.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

## 1.13 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <summary>
    /// <para>
    /// Represents the frequencies cache based link to its frequency number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{Doublet{TLink}, TLink}"/>
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
    ↪  IConverter<Doublet<TLink>, TLink>
    {
        /// <summary>
        /// <para>
        /// The cache.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly LinkFrequenciesCache<TLink> _cache;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪  cref="FrequenciesCacheBasedLinkToItsFrequencyNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cache">
        /// <para>A cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public
        ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
        ↪  cache) => _cache = cache;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
    }
}
```

## 1.14 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOf

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the marked sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceSymbolFrequencyOneOffCounter{TLink}"/>
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
    {
        /// <summary>
        /// <para>
        /// The marked sequence matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="MarkedSequenceSymbolFrequencyOneOffCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceLink">
        /// <para>A sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLink Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

## 1.15 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounte

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLink}"/>
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ILinks<TLink> _links;
        /// <summary>
        /// <para>
        /// The sequence link.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLink _sequenceLink;
        /// <summary>
        /// <para>
        /// The symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLink _symbol;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
        /// <para></para>
        /// </summary>
        protected TLink _total;

        /// <summary>
```

```
        /// <para>
        /// Initializes a new <see cref="SequenceSymbolFrequencyOneOffCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceLink">
        /// <para>A sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
        ↪  TLink symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪  IsElement, VisitElement);
            return _total;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="x">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
        ↪  IsPartialPoint

        /// <summary>
        /// <para>
        /// Determines whether this instance visit element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

## 1.16 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total marked sequence symbol frequency counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLink, TLink}"/>
    public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<TLink> _links;
        /// <summary>
        /// <para>
        /// The marked sequence matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

        /// <summary>
        /// <para>
        /// Counts the argument.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink argument) => new
        ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
        ↪  _markedSequenceMatcher, argument).Count();
    }
}
```

## 1.17 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyO

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total marked sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="TotalSequenceSymbolFrequencyOneOffCounter{TLink}"/>
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
    {
        /// <summary>
        /// <para>
        /// The marked sequence matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyOneOffCounter"/>
        ↪  instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
        ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
            : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Counts the sequence symbol frequency using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void CountSequenceSymbolFrequency(TLink link)
        {
```

```
            var symbolFrequencyCounter = new
            ↪  MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  _markedSequenceMatcher, link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
    }
}
```

## 1.18 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total sequence symbol frequency counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLink, TLink}"/>
    public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<TLink> _links;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalSequenceSymbolFrequencyCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;

        /// <summary>
        /// <para>
        /// Counts the symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="symbol">
        /// <para>The symbol.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count(TLink symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
    }
}
```

## 1.19 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffC

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
```

```
12      /// Represents the total sequence symbol frequency one off counter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="ICounter{TLink}"/>
17      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
18      {
19          /// <summary>
20          /// <para>
21          /// The default.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
26          /// <summary>
27          /// <para>
28          /// The default.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
33
34          /// <summary>
35          /// <para>
36          /// The links.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          protected readonly ILinks<TLink> _links;
41          /// <summary>
42          /// <para>
43          /// The symbol.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          protected readonly TLink _symbol;
48          /// <summary>
49          /// <para>
50          /// The visits.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          protected readonly HashSet<TLink> _visits;
55          /// <summary>
56          /// <para>
57          /// The total.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          protected TLink _total;
62
63          /// <summary>
64          /// <para>
65          /// Initializes a new <see cref="TotalSequenceSymbolFrequencyOneOffCounter"/> instance.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          /// <param name="links">
70          /// <para>A links.</para>
71          /// <para></para>
72          /// </param>
73          /// <param name="symbol">
74          /// <para>A symbol.</para>
75          /// <para></para>
76          /// </param>
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
79          {
80              _links = links;
81              _symbol = symbol;
82              _visits = new HashSet<TLink>();
83              _total = default;
84          }
85
86          /// <summary>
87          /// <para>
88          /// Counts this instance.
89          /// </para>
```

```
90          /// <para></para>
91          /// </summary>
92          /// <returns>
93          /// <para>The total.</para>
94          /// <para></para>
95          /// </returns>
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public TLink Count()
98          {
99              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
100             {
101                 return _total;
102             }
103             CountCore(_symbol);
104             return _total;
105         }
106
107         /// <summary>
108         /// <para>
109         /// Counts the core using the specified link.
110         /// </para>
111         /// <para></para>
112         /// </summary>
113         /// <param name="link">
114         /// <para>The link.</para>
115         /// <para></para>
116         /// </param>
117         [MethodImpl(MethodImplOptions.AggressiveInlining)]
118         private void CountCore(TLink link)
119         {
120             var any = _links.Constants.Any;
121             if (_equalityComparer.Equals(_links.Count(any, link), default))
122             {
123                 CountSequenceSymbolFrequency(link);
124             }
125             else
126             {
127                 _links.Each(EachElementHandler, any, link);
128             }
129         }
130
131         /// <summary>
132         /// <para>
133         /// Counts the sequence symbol frequency using the specified link.
134         /// </para>
135         /// <para></para>
136         /// </summary>
137         /// <param name="link">
138         /// <para>The link.</para>
139         /// <para></para>
140         /// </param>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         protected virtual void CountSequenceSymbolFrequency(TLink link)
143         {
144             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  link, _symbol);
145             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
146         }
147
148         /// <summary>
149         /// <para>
150         /// Eaches the element handler using the specified doublet.
151         /// </para>
152         /// <para></para>
153         /// </summary>
154         /// <param name="doublet">
155         /// <para>The doublet.</para>
156         /// <para></para>
157         /// </param>
158         /// <returns>
159         /// <para>The link</para>
160         /// <para></para>
161         /// </returns>
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         private TLink EachElementHandler(IList<TLink> doublet)
164         {
165             var constants = _links.Constants;
166             var doubletIndex = doublet[constants.IndexPart];
```

```
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

## 1.20 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/CachedSequenceHeightProvider.cs

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Represents the cached sequence height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceHeightProvider{TLink}"/>
    public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The height property marker.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink _heightPropertyMarker;
        /// <summary>
        /// <para>
        /// The base height provider.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        /// <summary>
        /// <para>
        /// The address to unary number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        /// <summary>
        /// <para>
        /// The unary number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        /// <summary>
        /// <para>
        /// The property operator.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IProperties<TLink, TLink, TLink> _propertyOperator;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachedSequenceHeightProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="baseHeightProvider">
        /// <para>A base height provider.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToUnaryNumberConverter">
        /// <para>A address to unary number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberToAddressConverter">
        /// <para>A unary number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="heightPropertyMarker">
        /// <para>A height property marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyOperator">
        /// <para>A property operator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        /// <summary>
        /// <para>
        /// Gets the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The height.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink sequence)
        {
            TLink height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

## 1.21 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
```

```
    /// <para>
    /// Represents the default sequence right height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ISequenceHeightProvider{TLink}"/>
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceHeightProvider<TLink>
    {
        /// <summary>
        /// <para>
        /// The element matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _elementMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceRightHeightProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="elementMatcher">
        /// <para>A element matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪  elementMatcher) : base(links) => _elementMatcher = elementMatcher;

        /// <summary>
        /// <para>
        /// Gets the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The height.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Get(TLink sequence)
        {
            var height = default(TLink);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
                pairOrElement = _links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

## 1.22 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/ISequenceHeightProvider.cs

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Defines the sequence height provider.
    /// </para>
    /// <para></para>
    /// </summary>
```

```
    /// <seealso cref="IProvider{TLink, TLink}"/>
    public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
    {
    }
}
```

## 1.23 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/FrequencyIncrementer.cs

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

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

namespace Platform.Data.Doublets.Incrementers
{
    /// <summary>
    /// <para>
    /// Represents the frequency incrementer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IIncrementer{TLink}"/>
    public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The frequency marker.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink _frequencyMarker;
        /// <summary>
        /// <para>
        /// The unary one.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink _unaryOne;
        /// <summary>
        /// <para>
        /// The unary number incrementer.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="FrequencyIncrementer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyMarker">
        /// <para>A frequency marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>A unary one.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberIncrementer">
        /// <para>A unary number incrementer.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
72          public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
            ↪  IIncrementer<TLink> unaryNumberIncrementer)
73              : base(links)
74          {
75              _frequencyMarker = frequencyMarker;
76              _unaryOne = unaryOne;
77              _unaryNumberIncrementer = unaryNumberIncrementer;
78          }
79
80          /// <summary>
81          /// <para>
82          /// Increments the frequency.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          /// <param name="frequency">
87          /// <para>The frequency.</para>
88          /// <para></para>
89          /// </param>
90          /// <returns>
91          /// <para>The link</para>
92          /// <para></para>
93          /// </returns>
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          public TLink Increment(TLink frequency)
96          {
97              var links = _links;
98              if (_equalityComparer.Equals(frequency, default))
99              {
100                 return links.GetOrCreate(_unaryOne, _frequencyMarker);
101             }
102             var incrementedSource =
                ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
103             return links.GetOrCreate(incrementedSource, _frequencyMarker);
104         }
105     }
106 }
```

## 1.24 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Incrementers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Incrementers
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the unary number incrementer.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksOperatorBase{TLink}"/>
16     /// <seealso cref="IIncrementer{TLink}"/>
17     public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
18     {
19         /// <summary>
20         /// <para>
21         /// The default.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪  EqualityComparer<TLink>.Default;
26
27         /// <summary>
28         /// <para>
29         /// The unary one.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         private readonly TLink _unaryOne;
34
35         /// <summary>
36         /// <para>
37         /// Initializes a new <see cref="UnaryNumberIncrementer"/> instance.
38         /// </para>
39         /// <para></para>
```

```
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>A unary one.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
        ↪  _unaryOne = unaryOne;

        /// <summary>
        /// <para>
        /// Increments the unary number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="unaryNumber">
        /// <para>The unary number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink unaryNumber)
        {
            var links = _links;
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return links.GetOrCreate(_unaryOne, _unaryOne);
            }
            var source = links.GetSource(unaryNumber);
            var target = links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return links.GetOrCreate(unaryNumber, _unaryOne);
            }
            else
            {
                return links.GetOrCreate(source, Increment(target));
            }
        }
    }
}
```

## 1.25 ./csharp/Platform.Data.Doublets.Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the cached frequency incrementing sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The cache.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly LinkFrequenciesCache<TLink> _cache;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachedFrequencyIncrementingSequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cache">
        /// <para>A cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
        ↪  _cache = cache;

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is indexed with increment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexedWithIncrement(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
```

```
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is indexed.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

## 1.26 ./csharp/Platform.Data.Doublets.Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the frequency incrementing sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceIndex{TLink}"/>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
    ↪ ISequenceIndex<TLink>
    {
        /// <summary>
```

```
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The frequency property operator.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        /// <summary>
        /// <para>
        /// The frequency incrementer.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IIncrementer<TLink> _frequencyIncrementer;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="FrequencyIncrementingSequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyPropertyOperator">
        /// <para>A frequency property operator.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyIncrementer">
        /// <para>A frequency incrementer.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IProperty<TLink, TLink>
        ↪  frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _frequencyIncrementer = frequencyIncrementer;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
            }
            return indexed;
        }

```

```
96          /// <summary>
97          /// <para>
98          /// Determines whether this instance is indexed with increment.
99          /// </para>
100         /// <para></para>
101         /// </summary>
102         /// <param name="source">
103         /// <para>The source.</para>
104         /// <para></para>
105         /// </param>
106         /// <param name="target">
107         /// <para>The target.</para>
108         /// <para></para>
109         /// </param>
110         /// <returns>
111         /// <para>The indexed.</para>
112         /// <para></para>
113         /// </returns>
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         private bool IsIndexedWithIncrement(TLink source, TLink target)
116         {
117             var link = _links.SearchOrDefault(source, target);
118             var indexed = !_equalityComparer.Equals(link, default);
119             if (indexed)
120             {
121                 Increment(link);
122             }
123             return indexed;
124         }
125
126         /// <summary>
127         /// <para>
128         /// Increments the link.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         /// <param name="link">
133         /// <para>The link.</para>
134         /// <para></para>
135         /// </param>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         private void Increment(TLink link)
138         {
139             var previousFrequency = _frequencyPropertyOperator.Get(link);
140             var frequency = _frequencyIncrementer.Increment(previousFrequency);
141             _frequencyPropertyOperator.Set(link, frequency);
142         }
143     }
144 }
```

## 1.27 ./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.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.Data.Doublets.Sequences.Indexes
7   {
8       /// <summary>
9       /// <para>
10      /// Defines the sequence index.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public interface ISequenceIndex<TLink>
15      {
16          /// <summary>
17          /// Индексирует последовательность глобально, и возвращает значение,
18          /// определяющие была ли запрошенная последовательность проиндексирована ранее.
19          /// </summary>
20          /// <param name="sequence">Последовательность для индексации.</param>
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          bool Add(IList<TLink> sequence);
23
24          /// <summary>
25          /// <para>
26          /// Determines whether this instance might contain.
27          /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool MightContain(IList<TLink> sequence);
    }
}
```

## 1.28 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceIndex(ILinks<TLink> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            for (; i >= 1; i--)
```

```
            {
                _links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            return indexed;
        }
    }
}
```

## 1.29 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the synchronized sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ISynchronizedLinks<TLink> _links;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SynchronizedSequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
```

```

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.ExecuteReadOperation(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.ExecuteReadOperation(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

## 1.30 ./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
```

```
    /// <summary>
    /// <para>
    /// Represents the unindex.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class Unindex<TLink> : ISequenceIndex<TLink>
    {
        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLink> sequence) => false;

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLink> sequence) => true;
    }
}
```

## 1.31 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/DecimalToRationalConverter.cs

```
using System.Numerics;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using System.Globalization;
using Platform.Data.Doublets.Numbers.Raw;

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

namespace Platform.Data.Doublets.Numbers.Rational
{
    /// <summary>
    /// <para>
    /// Represents the decimal to rational converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLink}"/>
    /// <seealso cref="IConverter{decimal, TLink}"/>
    public class DecimalToRationalConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪  IConverter<decimal, TLink>
        where TLink: struct
    {
        /// <summary>
        /// <para>
        /// The big integer to raw number sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly BigIntegerToRawNumberSequenceConverter<TLink>
        ↪  BigIntegerToRawNumberSequenceConverter;

        /// <summary>
        /// <para>
```

```
        /// Initializes a new <see cref="DecimalToRationalConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="bigIntegerToRawNumberSequenceConverter">
        /// <para>A big integer to raw number sequence converter.</para>
        /// <para></para>
        /// </param>
        public DecimalToRationalConverter(ILinks<TLink> links,
        ↪  BigIntegerToRawNumberSequenceConverter<TLink>
        ↪  bigIntegerToRawNumberSequenceConverter) : base(links)
        {
            BigIntegerToRawNumberSequenceConverter = bigIntegerToRawNumberSequenceConverter;
        }

        /// <summary>
        /// <para>
        /// Converts the decimal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@decimal">
        /// <para>The decimal.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public TLink Convert(decimal @decimal)
        {
            var decimalAsString = @decimal.ToString(CultureInfo.InvariantCulture);
            var dotPosition = decimalAsString.IndexOf('.');
            var decimalWithoutDots = decimalAsString;
            int digitsAfterDot = 0;
            if (dotPosition != -1)
            {
                decimalWithoutDots = decimalWithoutDots.Remove(dotPosition, 1);
                digitsAfterDot = decimalAsString.Length - 1 - dotPosition;
            }
            BigInteger denominator = new(System.Math.Pow(10, digitsAfterDot));
            BigInteger numerator = BigInteger.Parse(decimalWithoutDots);
            BigInteger greatestCommonDivisor;
            do
            {
                greatestCommonDivisor = BigInteger.GreatestCommonDivisor(numerator, denominator);
                numerator /= greatestCommonDivisor;
                denominator /= greatestCommonDivisor;
            }
            while (greatestCommonDivisor > 1);
            var numeratorLink = BigIntegerToRawNumberSequenceConverter.Convert(numerator);
            var denominatorLink = BigIntegerToRawNumberSequenceConverter.Convert(denominator);
            return _links.GetOrCreate(numeratorLink, denominatorLink);
        }
    }
}
```

## 1.32 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/RationalToDecimalConverter.cs

```
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Numbers.Raw;

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

namespace Platform.Data.Doublets.Numbers.Rational
{
    /// <summary>
    /// <para>
    /// Represents the rational to decimal converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink, decimal}"/>
```

```
17      public class RationalToDecimalConverter<TLink> : LinksDecoratorBase<TLink>,
        ↪  IConverter<TLink, decimal>
18          where TLink: struct
19      {
20          /// <summary>
21          /// <para>
22          /// The raw number sequence to big integer converter.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          public readonly RawNumberSequenceToBigIntegerConverter<TLink>
            ↪  RawNumberSequenceToBigIntegerConverter;
27
28          /// <summary>
29          /// <para>
30          /// Initializes a new <see cref="RationalToDecimalConverter"/> instance.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          /// <param name="links">
35          /// <para>A links.</para>
36          /// <para></para>
37          /// </param>
38          /// <param name="rawNumberSequenceToBigIntegerConverter">
39          /// <para>A raw number sequence to big integer converter.</para>
40          /// <para></para>
41          /// </param>
42          public RationalToDecimalConverter(ILinks<TLink> links,
            ↪  RawNumberSequenceToBigIntegerConverter<TLink>
            ↪  rawNumberSequenceToBigIntegerConverter) : base(links)
43          {
44              RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
45          }
46
47          /// <summary>
48          /// <para>
49          /// Converts the rational number.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          /// <param name="rationalNumber">
54          /// <para>The rational number.</para>
55          /// <para></para>
56          /// </param>
57          /// <returns>
58          /// <para>The decimal</para>
59          /// <para></para>
60          /// </returns>
61          public decimal Convert(TLink rationalNumber)
62          {
63              var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo⌋
                ↪  urce(rationalNumber));
64              var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get⌋
                ↪  Target(rationalNumber));
65              return numerator / denominator;
66          }
67      }
68  }
```

## 1.33 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs

```
1   using System.Collections.Generic;
2   using System.Numerics;
3   using Platform.Converters;
4   using Platform.Data.Doublets.Decorators;
5   using Platform.Numbers;
6   using Platform.Reflection;
7   using Platform.Unsafe;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data.Doublets.Numbers.Raw
12  {
13      /// <summary>
14      /// <para>
15      /// Represents the big integer to raw number sequence converter.
16      /// </para>
17      /// <para></para>
18      /// </summary>
19      /// <seealso cref="LinksDecoratorBase{TLink}"/>
```

```
    /// <seealso cref="IConverter{BigInteger, TLink}"/>
    public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪ IConverter<BigInteger, TLink>
        where TLink : struct
    {
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLink MaximumValue = NumericType<TLink>.MaxValue;
        /// <summary>
        /// <para>
        /// The maximum value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLink BitMask = Bit.ShiftRight(MaximumValue, 1);
        /// <summary>
        /// <para>
        /// The address to number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLink> AddressToNumberConverter;
        /// <summary>
        /// <para>
        /// The list to sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<IList<TLink>, TLink> ListToSequenceConverter;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLink NegativeNumberMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BigIntegerToRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceConverter">
        /// <para>A list to sequence converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="negativeNumberMarker">
        /// <para>A negative number marker.</para>
        /// <para></para>
        /// </param>
        public BigIntegerToRawNumberSequenceConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, IConverter<IList<TLink>,TLink> listToSequenceConverter,
        ↪ TLink negativeNumberMarker) : base(links)
        {
            AddressToNumberConverter = addressToNumberConverter;
            ListToSequenceConverter = listToSequenceConverter;
            NegativeNumberMarker = negativeNumberMarker;
        }

        /// <summary>
        /// <para>
        /// Gets the raw number parts using the specified big integer.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
95          /// <param name="bigInteger">
96          /// <para>The big integer.</para>
97          /// <para></para>
98          /// </param>
99          /// <returns>
100         /// <para>The raw numbers.</para>
101         /// <para></para>
102         /// </returns>
103         private List<TLink> GetRawNumberParts(BigInteger bigInteger)
104         {
105             List<TLink> rawNumbers = new();
106             BigInteger currentBigInt = bigInteger;
107             do
108             {
109                 var bigIntBytes = currentBigInt.ToByteArray();
110                 var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLink>(), BitMask);
111                 var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
112                 rawNumbers.Add(rawNumber);
113                 currentBigInt >>= 63;
114             }
115             while (currentBigInt > 0);
116             return rawNumbers;
117         }
118
119         /// <summary>
120         /// <para>
121         /// Converts the big integer.
122         /// </para>
123         /// <para></para>
124         /// </summary>
125         /// <param name="bigInteger">
126         /// <para>The big integer.</para>
127         /// <para></para>
128         /// </param>
129         /// <returns>
130         /// <para>The link</para>
131         /// <para></para>
132         /// </returns>
133         public TLink Convert(BigInteger bigInteger)
134         {
135             var sign = bigInteger.Sign;
136             var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
                ↪ bigInteger);
137             var numberSequence = ListToSequenceConverter.Convert(number);
138             return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
                ↪ numberSequence;
139         }
140     }
141 }
```

## 1.34 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/LongRawNumberSequenceToNumberConverter.c

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Stacks;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Numbers.Raw
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the long raw number sequence to number converter.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     /// <seealso cref="LinksDecoratorBase{TSource}"/>
20     /// <seealso cref="IConverter{TSource, TTarget}"/>
21     public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
       ↪ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
22     {
23         /// <summary>
24         /// <para>
25         /// The bits size.
26         /// </para>
27         /// <para></para>
```

```
28          /// </summary>
29          private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
30          /// <summary>
31          /// <para>
32          /// The default.
33          /// </para>
34          /// <para></para>
35          /// </summary>
36          private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
            ↪  UncheckedConverter<TSource, TTarget>.Default;
37
38          /// <summary>
39          /// <para>
40          /// The number to address converter.
41          /// </para>
42          /// <para></para>
43          /// </summary>
44          private readonly IConverter<TSource> _numberToAddressConverter;
45
46          /// <summary>
47          /// <para>
48          /// Initializes a new <see cref="LongRawNumberSequenceToNumberConverter"/> instance.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          /// <param name="links">
53          /// <para>A links.</para>
54          /// <para></para>
55          /// </param>
56          /// <param name="numberToAddressConverter">
57          /// <para>A number to address converter.</para>
58          /// <para></para>
59          /// </param>
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
            ↪  numberToAddressConverter) : base(links) => _numberToAddressConverter =
            ↪  numberToAddressConverter;
62
63          /// <summary>
64          /// <para>
65          /// Converts the source.
66          /// </para>
67          /// <para></para>
68          /// </summary>
69          /// <param name="source">
70          /// <para>The source.</para>
71          /// <para></para>
72          /// </param>
73          /// <returns>
74          /// <para>The target</para>
75          /// <para></para>
76          /// </returns>
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public TTarget Convert(TSource source)
79          {
80              var constants = Links.Constants;
81              var externalReferencesRange = constants.ExternalReferencesRange;
82              if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(source))
83              {
84                  return
                    ↪  _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
85              }
86              else
87              {
88                  var pair = Links.GetLink(source);
89                  var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
                    ↪  (link) => externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(link));
90                  TTarget result = default;
91                  foreach (var element in walker.Walk(source))
92                  {
93                      result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
94                  }
95                  return result;
96              }
97          }
98      }
```

```
}
```

## 1.35 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/NumberToLongRawNumberSequenceConverter.c

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Data.Doublets.Decorators;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the number to long raw number sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TTarget}"/>
    /// <seealso cref="IConverter{TSource, TTarget}"/>
    public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
    ↪  LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
        /// <summary>
        /// <para>
        /// The bits size.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
        /// <summary>
        /// <para>
        /// The bits size.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
        ↪  NumericType<TTarget>.BitsSize + 1);
        /// <summary>
        /// <para>
        /// The external zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TSource _maximumConvertableAddress = CheckedConverter<TTarget,
        ↪  TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪  UncheckedConverter<TSource, TTarget>.Default;

        /// <summary>
        /// <para>
        /// The address to number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TTarget> _addressToNumberConverter;

```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NumberToLongRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
        ↪  addressToNumberConverter) : base(links) => _addressToNumberConverter =
        ↪  addressToNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source)
        {
            if (_comparer.Compare(source, _maximumConvertableAddress) > 0)
            {
                var numberPart = Bit.And(source, _bitMask);
                var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
                ↪  .Convert(numberPart));
                return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
                ↪  _bitsPerRawNumber)));
            }
            else
            {
                return
                ↪  _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
            }
        }
    }
}
```

## 1.36 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/RawNumberSequenceToBigIntergerConverter.cs

```
using System;
using System.Collections.Generic;
using System.Numerics;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Unsafe;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the raw number sequence to big integer converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink, BigInteger}"/>
    public class RawNumberSequenceToBigIntegerConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪  IConverter<TLink, BigInteger>
        where TLink : struct
    {
```

```
25          /// <summary>
26          /// <para>
27          /// The default.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          public readonly EqualityComparer<TLink> EqualityComparer =
            ↪  EqualityComparer<TLink>.Default;
32          /// <summary>
33          /// <para>
34          /// The number to address converter.
35          /// </para>
36          /// <para></para>
37          /// </summary>
38          public readonly IConverter<TLink, TLink> NumberToAddressConverter;
39          /// <summary>
40          /// <para>
41          /// The left sequence walker.
42          /// </para>
43          /// <para></para>
44          /// </summary>
45          public readonly LeftSequenceWalker<TLink> LeftSequenceWalker;
46          /// <summary>
47          /// <para>
48          /// The negative number marker.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          public readonly TLink NegativeNumberMarker;
53
54          /// <summary>
55          /// <para>
56          /// Initializes a new <see cref="RawNumberSequenceToBigIntegerConverter"/> instance.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="links">
61          /// <para>A links.</para>
62          /// <para></para>
63          /// </param>
64          /// <param name="numberToAddressConverter">
65          /// <para>A number to address converter.</para>
66          /// <para></para>
67          /// </param>
68          /// <param name="negativeNumberMarker">
69          /// <para>A negative number marker.</para>
70          /// <para></para>
71          /// </param>
72          public RawNumberSequenceToBigIntegerConverter(ILinks<TLink> links, IConverter<TLink,
            ↪  TLink> numberToAddressConverter, TLink negativeNumberMarker) : base(links)
73          {
74              NumberToAddressConverter = numberToAddressConverter;
75              LeftSequenceWalker = new(links, new DefaultStack<TLink>());
76              NegativeNumberMarker = negativeNumberMarker;
77          }
78
79          /// <summary>
80          /// <para>
81          /// Converts the big integer.
82          /// </para>
83          /// <para></para>
84          /// </summary>
85          /// <param name="bigInteger">
86          /// <para>The big integer.</para>
87          /// <para></para>
88          /// </param>
89          /// <exception cref="Exception">
90          /// <para>Raw number sequence cannot be empty.</para>
91          /// <para></para>
92          /// </exception>
93          /// <returns>
94          /// <para>The big integer</para>
95          /// <para></para>
96          /// </returns>
97          public BigInteger Convert(TLink bigInteger)
98          {
99              var sign = 1;
100             var bigIntegerSequence = bigInteger;
```

```
            if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
            ↪  NegativeNumberMarker))
            {
                sign = -1;
                bigIntegerSequence = _links.GetTarget(bigInteger);
            }
            using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
            if (!enumerator.MoveNext())
            {
                throw new Exception("Raw number sequence cannot be empty.");
            }
            var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
            BigInteger currentBigInt = new(nextPart.ToBytes());
            while (enumerator.MoveNext())
            {
                currentBigInt <<= 63;
                nextPart = NumberToAddressConverter.Convert(enumerator.Current);
                currentBigInt |= new BigInteger(nextPart.ToBytes());
            }
            return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
        }
    }
}
```

## 1.37 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/AddressToUnaryNumberConverter.cs

```
using System.Collections.Generic;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the address to unary number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink}"/>
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TLink _zero = default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        /// <summary>
        /// <para>
        /// The power of to unary number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

        /// <summary>
        /// <para>
```

```
        /// Initializes a new <see cref="AddressToUnaryNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="powerOf2ToUnaryNumberConverter">
        /// <para>A power of to unary number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
        ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
        ↪  powerOf2ToUnaryNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink number)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
            ↪  NumericType<TLink>.BitsSize; i++)
            {
                if (_equalityComparer.Equals(Bit.And(number, _one), _one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

## 1.38 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the link to its frequency number conveter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{Doublet{TLink}, TLink}"/>
    public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<Doublet<TLink>, TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The frequency property operator.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        /// <summary>
        /// <para>
        /// The unary number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkToItsFrequencyNumberConveter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyPropertyOperator">
        /// <para>A frequency property operator.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberToAddressConverter">
        /// <para>A unary number to address converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkToItsFrequencyNumberConveter(
            ILinks<TLink> links,
            IProperty<TLink, TLink> frequencyPropertyOperator,
            IConverter<TLink> unaryNumberToAddressConverter)
            : base(links)
        {
            _frequencyPropertyOperator = frequencyPropertyOperator;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
        }

        /// <summary>
        /// <para>
        /// Converts the doublet.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="doublet">
        /// <para>The doublet.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentException">
        /// <para>Link ({doublet}) not found. </para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(Doublet<TLink> doublet)
        {
            var links = _links;
            var link = links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
```

```
            {
                return default;
            }
            var frequencyNumber = links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

## 1.39 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Converters;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the power of to unary number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{int, TLink}"/>
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<int, TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The unary number powers of.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink[] _unaryNumberPowersOf2;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="PowerOf2ToUnaryNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="one">
        /// <para>A one.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
        {
            _unaryNumberPowersOf2 = new TLink[64];
            _unaryNumberPowersOf2[0] = one;
        }

        /// <summary>
        /// <para>
        /// Converts the power.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="power">
        /// <para>The power.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The power of.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
            var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
            _unaryNumberPowersOf2[power] = powerOf2;
            return powerOf2;
        }
    }
}
```

## 1.40 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressAddOperationConverte

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address add operation converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink}"/>
    public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>, IConverter<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TLink _zero = default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        /// <summary>
        /// <para>
        /// The unary to int 64.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly Dictionary<TLink, TLink> _unaryToUInt64;
        /// <summary>
        /// <para>
        /// The unary one.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink _unaryOne;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnaryNumberToAddressAddOperationConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>A unary one.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
        }

        /// <summary>
        /// <para>
        /// Converts the unary number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="unaryNumber">
        /// <para>The unary number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, default))
            {
                return default;
            }
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return _one;
            }
            var links = _links;
            var source = links.GetSource(unaryNumber);
            var target = links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return _unaryToUInt64[unaryNumber];
            }
            else
            {
                var result = _unaryToUInt64[source];
                TLink lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = links.GetSource(target);
```

```
                    result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
                    target = links.GetTarget(target);
                }
                result = Arithmetic<TLink>.Add(result, lastValue);
                return result;
            }
        }

        /// <summary>
        /// <para>
        /// Creates the unary to u int 64 dictionary using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>The unary one.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The unary to int 64.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↪  links, TLink unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { unaryOne, _one }
            };
            var unary = unaryOne;
            var number = _one;
            for (var i = 1; i < 64; i++)
            {
                unary = links.GetOrCreate(unary, unary);
                number = Double(number);
                unaryToUInt64.Add(unary, number);
            }
            return unaryToUInt64;
        }

        /// <summary>
        /// <para>
        /// Doubles the number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) =>
        ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

## 1.41 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressOrOperationConverter

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address or operation converter.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink}"/>
    public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TLink _zero = default;
        /// <summary>
        /// <para>
        /// The zero.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        /// <summary>
        /// <para>
        /// The unary number power of indicies.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnaryNumberToAddressOrOperationConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="powerOf2ToUnaryNumberConverter">
        /// <para>A power of to unary number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
        ↪  TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
        ↪  = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);

        /// <summary>
        /// <para>
        /// Converts the source number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourceNumber">
        /// <para>The source number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink sourceNumber)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var source = sourceNumber;
```

```
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = links.GetTarget(source);
                    }
                }
            }
            return target;
        }

        /// <summary>
        /// <para>
        /// Creates the unary number power of 2 indicies dictionary using the specified power of 2 to unary number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="powerOf2ToUnaryNumberConverter">
        /// <para>The power of to unary number converter.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The unary number power of indicies.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, int> CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink> powerOf2ToUnaryNumberConverter)
        {
            var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
            for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
            return unaryNumberPowerOf2Indicies;
        }

        /// <summary>
        /// <para>
        /// Sets the bit using the specified target.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="powerOf2Index">
        /// <para>The power of index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target = Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

## 1.42 ./csharp/Platform.Data.Doublets.Sequences/Sequences.Experiments.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Collections.Sets;
using Platform.Collections.Stacks;
using Platform.Data.Exceptions;
```

```
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;
using Stack = System.Collections.Generic.Stack<ulong>;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the sequences.
    /// </para>
    /// <para></para>
    /// </summary>
    partial class Sequences
    {
        #region Create All Variants (Not Practical)

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
            });
        }

        /// <summary>
        /// <para>
        /// Creates the all variants 2 core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="startAt">
        /// <para>The start at.</para>
        /// <para></para>
        /// </param>
        /// <param name="stopAt">
        /// <para>The stop at.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para>Creation cancellation is not implemented.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The variants.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
        {
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
```

```
            var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
                        if (variant == Constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

        /// <summary>
        /// <para>
        /// Creates the all variants 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }

        /// <summary>
        /// <para>
        /// Creates the all variants 1 core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="results">
        /// <para>The results.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para>Creation cancellation is not implemented.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="NotImplementedException">
        /// <para>Creation cancellation is not implemented.</para>
        /// <para></para>
```

```
        /// </exception>
        /// <returns>
        /// <para>The results.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                results.Add(link);
                return results;
            }
            var innerSequenceLength = sequence.Length - 1;
            var innerSequence = new ulong[innerSequenceLength];
            for (var li = 0; li < innerSequenceLength; li++)
            {
                var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                for (var isi = 0; isi < li; isi++)
                {
                    innerSequence[isi] = sequence[isi];
                }
                innerSequence[li] = link;
                for (var isi = li + 1; isi < innerSequenceLength; isi++)
                {
                    innerSequence[isi] = sequence[isi + 1];
                }
                CreateAllVariants1Core(innerSequence, results);
            }
            return results;
        }

        #endregion

        /// <summary>
        /// <para>
        /// Eaches the 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The visited links.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> Each1(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }

        /// <summary>
        /// <para>
        /// Eaches the 1 using the specified handler.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(doublet =>
                    {
                        if (innerSequence == null)
                        {
                            innerSequence = new ulong[innerSequenceLength];
                            for (var isi = 0; isi < linkIndex; isi++)
                            {
                                innerSequence[isi] = sequence[isi];
                            }
                            for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                            {
                                innerSequence[isi] = sequence[isi + 1];
                            }
                        }
                        innerSequence[linkIndex] = doublet[Constants.IndexPart];
                        Each1(handler, innerSequence);
                        return Constants.Continue;
                    }, Constants.Any, left, right);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Eaches the part using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The visited links.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> EachPart(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                }
                return Constants.Continue;
```

```
            }, sequence);
            return visitedLinks;
        }

        /// <summary>
        /// <para>
        /// Eaches the part using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                    return handler(new LinkAddress<LinkIndex>(linkIndex));
                }
                return Constants.Continue;
            }, sequence);
        }

        /// <summary>
        /// <para>
        /// Eaches the part core using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
        ↪ sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return;
            }
            Links.EnsureLinkIsAnyOrExists(sequence);
            if (sequence.Length == 1)
            {
                var link = sequence[0];
                if (link > 0)
                {
                    handler(new LinkAddress<LinkIndex>(link));
                }
                else
                {
                    Links.Each(Constants.Any, Constants.Any, handler);
                }
            }
            else if (sequence.Length == 2)
            {
                //_links.Each(sequence[0], sequence[1], handler);
                //  o_|      x_o ...
                // x_|        |___|
```

```
                Links.Each(sequence[1], Constants.Any, doublet =>
                {
                    var match = Links.SearchOrDefault(sequence[0], doublet);
                    if (match != Constants.Null)
                    {
                        handler(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                // |_x      ... x_o
                //  |_o      |___|
                Links.Each(Constants.Any, sequence[0], doublet =>
                {
                    var match = Links.SearchOrDefault(doublet, sequence[1]);
                    if (match != 0)
                    {
                        handler(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                //           ._x o_.
                //            |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                throw new NotImplementedException();
            }
        }

        /// <summary>
        /// <para>
        /// Partials the step right using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, left, doublet =>
            {
                StepRight(handler, doublet, right);
                if (left != doublet)
                {
                    PartialStepRight(handler, doublet, right);
                }
                return true;
            });
        }

        /// <summary>
        /// <para>
        /// Steps the right using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
```

```
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(left, Constants.Any, rightStep =>
            {
                TryStepRightUp(handler, right, rightStep);
                return true;
            });
        }

        /// <summary>
        /// <para>
        /// Tries the step right up using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="stepFrom">
        /// <para>The step from.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
        ↪  stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        // TODO: Test
        /// <summary>
        /// <para>
        /// Partials the step left using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(right, Constants.Any, doublet =>
            {
                StepLeft(handler, left, doublet);
                if (right != doublet)
                {
                    PartialStepLeft(handler, left, doublet);
                }
                return true;
            });
```

```
        }

        /// <summary>
        /// <para>
        /// Steps the left using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, right, leftStep =>
            {
                TryStepLeftUp(handler, left, leftStep);
                return true;
            });
        }

        /// <summary>
        /// <para>
        /// Tries the step left up using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="stepFrom">
        /// <para>The step from.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance starts with.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool StartsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var firstSource = Links.Unsync.GetSource(upStep);
            while (firstSource != link && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            return firstSource == link;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance ends with.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool EndsWith(ulong sequence, ulong link)
        {
            var upStep = sequence;
            var lastTarget = Links.Unsync.GetTarget(upStep);
            while (lastTarget != link && lastTarget != upStep)
            {
                upStep = lastTarget;
                lastTarget = Links.Unsync.GetTarget(upStep);
            }
            return lastTarget == link;
        }

        /// <summary>
        /// <para>
        /// Gets the all matching sequences 0 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new List<ulong>();
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
```

```
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var linksInSequence = new HashSet<ulong>(sequence);
                    void handler(IList<LinkIndex> result)
                    {
                        var resultIndex = result[Links.Constants.IndexPart];
                        var filterPosition = 0;
                        StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
                            {
                                if (filterPosition == sequence.Length)
                                {
                                    filterPosition = -2; // Длиннее чем нужно
                                    return false;
                                }
                                if (x != sequence[filterPosition])
                                {
                                    filterPosition = -1;
                                    return false; // Начинается иначе
                                }
                                filterPosition++;

                                return true;
                            });
                        if (filterPosition == sequence.Length)
                        {
                            results.Add(resultIndex);
                        }
                    }
                    if (sequence.Length >= 2)
                    {
                        StepRight(handler, sequence[0], sequence[1]);
                    }
                    var last = sequence.Length - 2;
                    for (var i = 1; i < last; i++)
                    {
                        PartialStepRight(handler, sequence[i], sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(handler, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all matching sequences 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (sequence.Length > 0)
                {
```

```
                    Links.EnsureLinkExists(sequence);
                    var firstElement = sequence[0];
                    if (sequence.Length == 1)
                    {
                        results.Add(firstElement);
                        return results;
                    }
                    if (sequence.Length == 2)
                    {
                        var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
                        if (doublet != Constants.Null)
                        {
                            results.Add(doublet);
                        }
                        return results;
                    }
                    var matcher = new Matcher(this, sequence, results, null);
                    if (sequence.Length >= 2)
                    {
                        StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
                    }
                    var last = sequence.Length - 2;
                    for (var i = 1; i < last; i++)
                    {
                        PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
                        ↪  sequence[i + 1]);
                    }
                    if (sequence.Length >= 3)
                    {
                        StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// The max sequence format size.
        /// </para>
        /// <para></para>
        /// </summary>
        public const int MaxSequenceFormatSize = 200;

        /// <summary>
        /// <para>
        /// Formats the sequence using the specified sequence link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceLink">
        /// <para>The sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="knownElements">
        /// <para>The known elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
        ↪  => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

        /// <summary>
        /// <para>
        /// Formats the sequence using the specified sequence link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceLink">
        /// <para>The sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="elementToString">
```

```
857         /// <para>The element to string.</para>
858         /// <para></para>
859         /// </param>
860         /// <param name="insertComma">
861         /// <para>The insert comma.</para>
862         /// <para></para>
863         /// </param>
864         /// <param name="knownElements">
865         /// <para>The known elements.</para>
866         /// <para></para>
867         /// </param>
868         /// <returns>
869         /// <para>The string</para>
870         /// <para></para>
871         /// </returns>
872         [MethodImpl(MethodImplOptions.AggressiveInlining)]
873         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
              ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
              ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
              ↪ elementToString, insertComma, knownElements));
874
875         /// <summary>
876         /// <para>
877         /// Formats the sequence using the specified links.
878         /// </para>
879         /// <para></para>
880         /// </summary>
881         /// <param name="links">
882         /// <para>The links.</para>
883         /// <para></para>
884         /// </param>
885         /// <param name="sequenceLink">
886         /// <para>The sequence link.</para>
887         /// <para></para>
888         /// </param>
889         /// <param name="elementToString">
890         /// <para>The element to string.</para>
891         /// <para></para>
892         /// </param>
893         /// <param name="insertComma">
894         /// <para>The insert comma.</para>
895         /// <para></para>
896         /// </param>
897         /// <param name="knownElements">
898         /// <para>The known elements.</para>
899         /// <para></para>
900         /// </param>
901         /// <returns>
902         /// <para>The string</para>
903         /// <para></para>
904         /// </returns>
905         [MethodImpl(MethodImplOptions.AggressiveInlining)]
906         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
              ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
              ↪ LinkIndex[] knownElements)
907         {
908             var linksInSequence = new HashSet<ulong>(knownElements);
909             //var entered = new HashSet<ulong>();
910             var sb = new StringBuilder();
911             sb.Append('{');
912             if (links.Exists(sequenceLink))
913             {
914                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
915                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
                          ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
916                     {
917                         if (insertComma && sb.Length > 1)
918                         {
919                             sb.Append(',');
920                         }
921                         //if (entered.Contains(element))
922                         //{
923                         //    sb.Append('{');
924                         //    elementToString(sb, element);
925                         //    sb.Append('}');
926                         //}
927                         //else
```

```
                    elementToString(sb, element);
                    if (sb.Length < MaxSequenceFormatSize)
                    {
                        return true;
                    }
                    sb.Append(insertComma ? ", ..." : "...");
                    return false;
                });
        }
        sb.Append('}');
        return sb.ToString();
    }

    /// <summary>
    /// <para>
    /// Safes the format sequence using the specified sequence link.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="sequenceLink">
    /// <para>The sequence link.</para>
    /// <para></para>
    /// </param>
    /// <param name="knownElements">
    /// <para>The known elements.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The string</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪  knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪  knownElements);

    /// <summary>
    /// <para>
    /// Safes the format sequence using the specified sequence link.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="sequenceLink">
    /// <para>The sequence link.</para>
    /// <para></para>
    /// </param>
    /// <param name="elementToString">
    /// <para>The element to string.</para>
    /// <para></para>
    /// </param>
    /// <param name="insertComma">
    /// <para>The insert comma.</para>
    /// <para></para>
    /// </param>
    /// <param name="knownElements">
    /// <para>The known elements.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The string</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪  Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪  sequenceLink, elementToString, insertComma, knownElements));

    /// <summary>
    /// <para>
    /// Safes the format sequence using the specified links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="links">
    /// <para>The links.</para>
    /// <para></para>
    /// </param>
```

```
        /// <param name="sequenceLink">
        /// <para>The sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="elementToString">
        /// <para>The element to string.</para>
        /// <para></para>
        /// </param>
        /// <param name="insertComma">
        /// <para>The insert comma.</para>
        /// <para></para>
        /// </param>
        /// <param name="knownElements">
        /// <para>The known elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
        ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
        ↪  LinkIndex[] knownElements)
        {
            var linksInSequence = new HashSet<ulong>(knownElements);
            var entered = new HashSet<ulong>();
            var sb = new StringBuilder();
            sb.Append('{');
            if (links.Exists(sequenceLink))
            {
                StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
                    x => linksInSequence.Contains(x) || links.IsFullPoint(x),
                    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
                    {
                        if (insertComma && sb.Length > 1)
                        {
                            sb.Append(',');
                        }
                        if (entered.Contains(element))
                        {
                            sb.Append('{');
                            elementToString(sb, element);
                            sb.Append('}');
                        }
                        else
                        {
                            elementToString(sb, element);
                        }
                        if (sb.Length < MaxSequenceFormatSize)
                        {
                            return true;
                        }
                        sb.Append(insertComma ? ", ..." : "...");
                        return false;
                    });
            }
            sb.Append('}');
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences 0 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
        {
```

```
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
                    var filteredResults = new List<ulong>();
                    var linksInSequence = new HashSet<ulong>(sequence);
                    foreach (var result in results)
                    {
                        var filterPosition = -1;
                        StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
                            {
                                if (filterPosition == (sequence.Length - 1))
                                {
                                    return false;
                                }
                                if (filterPosition >= 0)
                                {
                                    if (x == sequence[filterPosition + 1])
                                    {
                                        filterPosition++;
                                    }
                                    else
                                    {
                                        return false;
                                    }
                                }
                                if (filterPosition < 0)
                                {
                                    if (x == sequence[0])
                                    {
                                        filterPosition = 0;
                                    }
                                }
                                return true;
                            });
                        if (filterPosition == (sequence.Length - 1))
                        {
                            filteredResults.Add(result);
                        }
                    }
                    return filteredResults;
                }
                return new List<ulong>();
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
```

```
                for (var i = 0; i < sequence.Length; i++)
                {
                    AllUsagesCore(sequence[i], results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

    /// <summary>
    /// <para>
    /// Determines whether this instance get all partially matching sequences 2.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="handler">
    /// <para>The handler.</para>
    /// <para></para>
    /// </param>
    /// <param name="sequence">
    /// <para>The sequence.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
    ↪  params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                Links.EnsureLinkExists(sequence);

                var results = new HashSet<ulong>();
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, handler);
                for (var i = 0; i < sequence.Length; i++)
                {
                    if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                    {
                        return false;
                    }
                }
                return true;
            }
            return true;
        });
    }

    //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    //{
    //    return Sync.ExecuteReadOperation(() =>
    //    {
    //        if (sequence.Length > 0)
    //        {
    //            _links.EnsureEachLinkIsAnyOrExists(sequence);

    //            var firstResults = new HashSet<ulong>();
    //            var lastResults = new HashSet<ulong>();

    //            var first = sequence.First(x => x != LinksConstants.Any);
    //            var last = sequence.Last(x => x != LinksConstants.Any);

    //            AllUsagesCore(first, firstResults);
    //            AllUsagesCore(last, lastResults);

    //            firstResults.IntersectWith(lastResults);

    //            //for (var i = 0; i < sequence.Length; i++)
    //            //    AllUsagesCore(sequence[i], results);

```

```
1230            //            var filteredResults = new HashSet<ulong>();
1231            //            var matcher = new Matcher(this, sequence, filteredResults, null);
1232            //            matcher.AddAllPartialMatchedToResults(firstResults);
1233            //            return filteredResults;
1234            //        }
1235
1236            //        return new HashSet<ulong>();
1237            //    });
1238            //}
1239
1240            /// <summary>
1241            /// <para>
1242            /// Gets the all partially matching sequences 3 using the specified sequence.
1243            /// </para>
1244            /// <para></para>
1245            /// </summary>
1246            /// <param name="sequence">
1247            /// <para>The sequence.</para>
1248            /// <para></para>
1249            /// </param>
1250            /// <returns>
1251            /// <para>A hash set of ulong</para>
1252            /// <para></para>
1253            /// </returns>
1254            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1255            public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
1256            {
1257                return _sync.ExecuteReadOperation(() =>
1258                {
1259                    if (sequence.Length > 0)
1260                    {
1261                        ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
1262                        var firstResults = new HashSet<ulong>();
1263                        var lastResults = new HashSet<ulong>();
1264                        var first = sequence.First(x => x != Constants.Any);
1265                        var last = sequence.Last(x => x != Constants.Any);
1266                        AllUsagesCore(first, firstResults);
1267                        AllUsagesCore(last, lastResults);
1268                        firstResults.IntersectWith(lastResults);
1269                        //for (var i = 0; i < sequence.Length; i++)
1270                        //    AllUsagesCore(sequence[i], results);
1271                        var filteredResults = new HashSet<ulong>();
1272                        var matcher = new Matcher(this, sequence, filteredResults, null);
1273                        matcher.AddAllPartialMatchedToResults(firstResults);
1274                        return filteredResults;
1275                    }
1276                    return new HashSet<ulong>();
1277                });
1278            }
1279
1280            /// <summary>
1281            /// <para>
1282            /// Gets the all partially matching sequences 4 using the specified read as elements.
1283            /// </para>
1284            /// <para></para>
1285            /// </summary>
1286            /// <param name="readAsElements">
1287            /// <para>The read as elements.</para>
1288            /// <para></para>
1289            /// </param>
1290            /// <param name="sequence">
1291            /// <para>The sequence.</para>
1292            /// <para></para>
1293            /// </param>
1294            /// <returns>
1295            /// <para>A hash set of ulong</para>
1296            /// <para></para>
1297            /// </returns>
1298            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1299            public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
                ↪ IList<ulong> sequence)
1300            {
1301                return _sync.ExecuteReadOperation(() =>
1302                {
1303                    if (sequence.Count > 0)
1304                    {
1305                        Links.EnsureLinkExists(sequence);
1306                        var results = new HashSet<LinkIndex>();
```

```
                    //var nextResults = new HashSet<ulong>();
                    //for (var i = 0; i < sequence.Length; i++)
                    //{
                    //    AllUsagesCore(sequence[i], nextResults);
                    //    if (results.IsNullOrEmpty())
                    //    {
                    //        results = nextResults;
                    //        nextResults = new HashSet<ulong>();
                    //    }
                    //    else
                    //    {
                    //        results.IntersectWith(nextResults);
                    //        nextResults.Clear();
                    //    }
                    //}
                    var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                    collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < sequence.Count; i++)
                    {
                        var collector = new AllUsagesCollector1(Links.Unsync, next);
                        collector.Collect(Links.Unsync.GetLink(sequence[i]));

                        results.IntersectWith(next);
                        next.Clear();
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new Matcher(this, sequence, filteredResults, null,
                    ↪  readAsElements);
                    matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
                    ↪  x)); // OrderBy is a Hack
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        // Does not work
        //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
        ↪  params ulong[] sequence)
        //{
        //    var visited = new HashSet<ulong>();
        //    var results = new HashSet<ulong>();
        //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
        ↪  true; }, readAsElements);
        //    var last = sequence.Length - 1;
        //    for (var i = 0; i < last; i++)
        //    {
        //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
        //    }
        //    return results;
        //}

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    //var firstElement = sequence[0];
                    //if (sequence.Length == 1)
                    //{
```

```
//    //results.Add(firstElement);
//    return results;
//}
//if (sequence.Length == 2)
//{
//    //var doublet = _links.SearchCore(firstElement, sequence[1]);
//    //if (doublet != Doublets.Links.Null)
//    //    results.Add(doublet);
//    return results;
//}
//var lastElement = sequence[sequence.Length - 1];
//Func<ulong, bool> handler = x =>
//{
//    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
↪  results.Add(x);
//    return true;
//};
//if (sequence.Length >= 2)
//    StepRight(handler, sequence[0], sequence[1]);
//var last = sequence.Length - 2;
//for (var i = 1; i < last; i++)
//    PartialStepRight(handler, sequence[i], sequence[i + 1]);
//if (sequence.Length >= 3)
//    StepLeft(handler, sequence[sequence.Length - 2],
↪  sequence[sequence.Length - 1]);
//////if (sequence.Length == 1)
//////{
//////    throw new NotImplementedException(); // all sequences, containing
↪  this element?
//////}
//////if (sequence.Length == 2)
//////{
//////    var results = new List<ulong>();
//////    PartialStepRight(results.Add, sequence[0], sequence[1]);
//////    return results;
//////}
//////var matches = new List<List<ulong>>();
//////var last = sequence.Length - 1;
//////for (var i = 0; i < last; i++)
//////{
//////    var results = new List<ulong>();
//////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
//////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
//////    if (results.Count > 0)
//////        matches.Add(results);
//////    else
//////        return results;
//////    if (matches.Count == 2)
//////    {
//////        var merged = new List<ulong>();
//////        for (var j = 0; j < matches[0].Count; j++)
//////            for (var k = 0; k < matches[1].Count; k++)
//////                CloseInnerConnections(merged.Add, matches[0][j],
↪  matches[1][k]);
//////        if (merged.Count > 0)
//////            matches = new List<List<ulong>> { merged };
//////        else
//////            return new List<ulong>();
//////    }
//////}
//////if (matches.Count > 0)
//////{
//////    var usages = new HashSet<ulong>();
//////    for (int i = 0; i < sequence.Length; i++)
//////    {
//////        AllUsagesCore(sequence[i], usages);
//////    }
//////    //for (int i = 0; i < matches[0].Count; i++)
//////    //    AllUsagesCore(matches[0][i], usages);
//////    //usages.UnionWith(matches[0]);
//////    return usages.ToList();
//////}
var firstLinkUsages = new HashSet<ulong>();
AllUsagesCore(sequence[0], firstLinkUsages);
firstLinkUsages.Add(sequence[0]);
//var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
↪  sequence[0] }; // or all sequences, containing this element?
```

```
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                    ↪ 1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                    ↪ firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

        /// <remarks>
        /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> AllUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var usages = new HashSet<ulong>();
                AllUsagesCore(link, usages);
                return usages;
            });
        }

        // При сборе всех использований (последовательностей) можно сохранять обратный путь к
        ↪ той связи с которой начинался поиск (STTTSSSTT),
        // причём достаточно одного бита для хранения перехода влево или вправо
        /// <summary>
        /// <para>
        /// Alls the usages core using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="usages">
        /// <para>The usages.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllUsagesCore(ulong link, HashSet<ulong> usages)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    AllUsagesCore(doublet, usages);
                }
                return true;
            }
            Links.Unsync.Each(link, Constants.Any, handler);
            Links.Unsync.Each(Constants.Any, link, handler);
        }

        /// <summary>
        /// <para>
        /// Alls the bottom usages using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> AllBottomUsages(ulong link)
        {
            return _sync.ExecuteReadOperation(() =>
```

```
            {
                var visits = new HashSet<ulong>();
                var usages = new HashSet<ulong>();
                AllBottomUsagesCore(link, visits, usages);
                return usages;
            });
        }

        /// <summary>
        /// <para>
        /// Alls the bottom usages core using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="visits">
        /// <para>The visits.</para>
        /// <para></para>
        /// </param>
        /// <param name="usages">
        /// <para>The usages.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
        ↪  usages)
        {
            bool handler(ulong doublet)
            {
                if (visits.Add(doublet))
                {
                    AllBottomUsagesCore(doublet, visits, usages);
                }
                return true;
            }
            if (Links.Unsync.Count(Constants.Any, link) == 0)
            {
                usages.Add(link);
            }
            else
            {
                Links.Unsync.Each(link, Constants.Any, handler);
                Links.Unsync.Each(Constants.Any, link, handler);
            }
        }

        /// <summary>
        /// <para>
        /// Calculates the total symbol frequency core using the specified symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="symbol">
        /// <para>The symbol.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
        {
            if (Options.UseSequenceMarker)
            {
                var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                ↪  Options.MarkedSequenceMatcher, symbol);
                return counter.Count();
            }
            else
            {
                var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
                ↪  symbol);
                return counter.Count();
            }
```

```
1603            }
1604
1605            /// <summary>
1606            /// <para>
1607            /// Determines whether this instance all usages core 1.
1608            /// </para>
1609            /// <para></para>
1610            /// </summary>
1611            /// <param name="link">
1612            /// <para>The link.</para>
1613            /// <para></para>
1614            /// </param>
1615            /// <param name="usages">
1616            /// <para>The usages.</para>
1617            /// <para></para>
1618            /// </param>
1619            /// <param name="outerHandler">
1620            /// <para>The outer handler.</para>
1621            /// <para></para>
1622            /// </param>
1623            /// <returns>
1624            /// <para>The bool</para>
1625            /// <para></para>
1626            /// </returns>
1627            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1628            private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
                ↪ LinkIndex> outerHandler)
1629            {
1630                bool handler(ulong doublet)
1631                {
1632                    if (usages.Add(doublet))
1633                    {
1634                        if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
1635                        {
1636                            return false;
1637                        }
1638                        if (!AllUsagesCore1(doublet, usages, outerHandler))
1639                        {
1640                            return false;
1641                        }
1642                    }
1643                    return true;
1644                }
1645                return Links.Unsync.Each(link, Constants.Any, handler)
1646                    && Links.Unsync.Each(Constants.Any, link, handler);
1647            }
1648
1649            /// <summary>
1650            /// <para>
1651            /// Calculates the all usages using the specified totals.
1652            /// </para>
1653            /// <para></para>
1654            /// </summary>
1655            /// <param name="totals">
1656            /// <para>The totals.</para>
1657            /// <para></para>
1658            /// </param>
1659            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1660            public void CalculateAllUsages(ulong[] totals)
1661            {
1662                var calculator = new AllUsagesCalculator(Links, totals);
1663                calculator.Calculate();
1664            }
1665
1666            /// <summary>
1667            /// <para>
1668            /// Calculates the all usages 2 using the specified totals.
1669            /// </para>
1670            /// <para></para>
1671            /// </summary>
1672            /// <param name="totals">
1673            /// <para>The totals.</para>
1674            /// <para></para>
1675            /// </param>
1676            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1677            public void CalculateAllUsages2(ulong[] totals)
1678            {
1679                var calculator = new AllUsagesCalculator2(Links, totals);
```

```
            calculator.Calculate();
        }

        /// <summary>
        /// <para>
        /// Represents the all usages calculator.
        /// </para>
        /// <para></para>
        /// </summary>
        private class AllUsagesCalculator
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly SynchronizedLinks<ulong> _links;
            /// <summary>
            /// <para>
            /// The totals.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ulong[] _totals;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="totals">
            /// <para>A totals.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }

            /// <summary>
            /// <para>
            /// Calculates this instance.
            /// </para>
            /// <para></para>
            /// </summary>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

            /// <summary>
            /// <para>
            /// Determines whether this instance calculate core.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool CalculateCore(ulong link)
            {
                if (_totals[link] == 0)
                {
                    var total = 1UL;
                    _totals[link] = total;
```

```
                    var visitedChildren = new HashSet<ulong>();
                    bool linkCalculator(ulong child)
                    {
                        if (link != child && visitedChildren.Add(child))
                        {
                            total += _totals[child] == 0 ? 1 : _totals[child];
                        }
                        return true;
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
                    _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
                    _totals[link] = total;
                }
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Represents the all usages calculator.
        /// </para>
        /// <para></para>
        /// </summary>
        private class AllUsagesCalculator2
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly SynchronizedLinks<ulong> _links;
            /// <summary>
            /// <para>
            /// The totals.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ulong[] _totals;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesCalculator2"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="totals">
            /// <para>A totals.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
            {
                _links = links;
                _totals = totals;
            }

            /// <summary>
            /// <para>
            /// Calculates this instance.
            /// </para>
            /// <para></para>
            /// </summary>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
            ↪  CalculateCore);

            /// <summary>
            /// <para>
            /// Determines whether this instance is element.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
```

```
1834                /// <para>The link.</para>
1835                /// <para></para>
1836                /// </param>
1837                /// <returns>
1838                /// <para>The bool</para>
1839                /// <para></para>
1840                /// </returns>
1841                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1842                private bool IsElement(ulong link)
1843                {
1844                    //_linksInSequence.Contains(link) ||
1845                    return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
                        ↪  link;
1846                }
1847
1848                /// <summary>
1849                /// <para>
1850                /// Determines whether this instance calculate core.
1851                /// </para>
1852                /// <para></para>
1853                /// </summary>
1854                /// <param name="link">
1855                /// <para>The link.</para>
1856                /// <para></para>
1857                /// </param>
1858                /// <returns>
1859                /// <para>The bool</para>
1860                /// <para></para>
1861                /// </returns>
1862                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1863                private bool CalculateCore(ulong link)
1864                {
1865                    // TODO: Проработать защиту от зацикливания
1866                    // Основано на SequenceWalker.WalkLeft
1867                    Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1868                    Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1869                    Func<ulong, bool> isElement = IsElement;
1870                    void visitLeaf(ulong parent)
1871                    {
1872                        if (link != parent)
1873                        {
1874                            _totals[parent]++;
1875                        }
1876                    }
1877                    void visitNode(ulong parent)
1878                    {
1879                        if (link != parent)
1880                        {
1881                            _totals[parent]++;
1882                        }
1883                    }
1884                    var stack = new Stack();
1885                    var element = link;
1886                    if (isElement(element))
1887                    {
1888                        visitLeaf(element);
1889                    }
1890                    else
1891                    {
1892                        while (true)
1893                        {
1894                            if (isElement(element))
1895                            {
1896                                if (stack.Count == 0)
1897                                {
1898                                    break;
1899                                }
1900                                element = stack.Pop();
1901                                var source = getSource(element);
1902                                var target = getTarget(element);
1903                                // Обработка элемента
1904                                if (isElement(target))
1905                                {
1906                                    visitLeaf(target);
1907                                }
1908                                if (isElement(source))
1909                                {
1910                                    visitLeaf(source);
```

```
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        visitNode(element);
                        element = getTarget(element);
                    }
                }
            }
            _totals[link]++;
            return true;
        }
    }

    /// <summary>
    /// <para>
    /// Represents the all usages collector.
    /// </para>
    /// <para></para>
    /// </summary>
    private class AllUsagesCollector
    {
        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<ulong> _links;
        /// <summary>
        /// <para>
        /// The usages.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly HashSet<ulong> _usages;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="AllUsagesCollector"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="usages">
        /// <para>A usages.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
        {
            _links = links;
            _usages = usages;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance collect.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Collect(ulong link)
        {
            if (_usages.Add(link))
            {
```

```
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Represents the all usages collector.
        /// </para>
        /// <para></para>
        /// </summary>
        private class AllUsagesCollector1
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ILinks<ulong> _links;
            /// <summary>
            /// <para>
            /// The usages.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly HashSet<ulong> _usages;
            /// <summary>
            /// <para>
            /// The continue.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ulong _continue;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesCollector1"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="usages">
            /// <para>A usages.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
                _continue = _links.Constants.Continue;
            }

            /// <summary>
            /// <para>
            /// Collects the link.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The continue.</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ulong Collect(IList<ulong> link)
            {
                var linkIndex = _links.GetIndex(link);
                if (_usages.Add(linkIndex))
                {
```

```
                    _links.Each(Collect, _links.Constants.Any, linkIndex);
                }
                return _continue;
            }
        }

        /// <summary>
        /// <para>
        /// Represents the all usages collector.
        /// </para>
        /// <para></para>
        /// </summary>
        private class AllUsagesCollector2
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ILinks<ulong> _links;
            /// <summary>
            /// <para>
            /// The usages.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly BitString _usages;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesCollector2"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="usages">
            /// <para>A usages.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
            {
                _links = links;
                _usages = usages;
            }

            /// <summary>
            /// <para>
            /// Determines whether this instance collect.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_usages.Add((long)link))
                {
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Represents the all usages intersecting collector.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        private class AllUsagesIntersectingCollector
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly SynchronizedLinks<ulong> _links;
            /// <summary>
            /// <para>
            /// The intersect with.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly HashSet<ulong> _intersectWith;
            /// <summary>
            /// <para>
            /// The usages.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly HashSet<ulong> _usages;
            /// <summary>
            /// <para>
            /// The enter.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly HashSet<ulong> _enter;

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesIntersectingCollector"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="intersectWith">
            /// <para>A intersect with.</para>
            /// <para></para>
            /// </param>
            /// <param name="usages">
            /// <para>A usages.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
            ↪  intersectWith, HashSet<ulong> usages)
            {
                _links = links;
                _intersectWith = intersectWith;
                _usages = usages;
                _enter = new HashSet<ulong>(); // защита от зацикливания
            }

            /// <summary>
            /// <para>
            /// Determines whether this instance collect.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
```

```
            {
                if (_enter.Add(link))
                {
                    if (_intersectWith.Contains(link))
                    {
                        _usages.Add(link);
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, Collect);
                    _links.Unsync.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Closes the inner connections using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }

        /// <summary>
        /// <para>
        /// Alls the close connections using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            // Direct
            if (left == right)
            {
                handler(new LinkAddress<LinkIndex>(left));
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != Constants.Null)
            {
                handler(new LinkAddress<LinkIndex>(doublet));
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
```

```
2301                PartialStepLeft(handler, left, right);
2302            }
2303
2304            /// <summary>
2305            /// <para>
2306            /// Gets the all partially matching sequences core using the specified sequence.
2307            /// </para>
2308            /// <para></para>
2309            /// </summary>
2310            /// <param name="sequence">
2311            /// <para>The sequence.</para>
2312            /// <para></para>
2313            /// </param>
2314            /// <param name="previousMatchings">
2315            /// <para>The previous matchings.</para>
2316            /// <para></para>
2317            /// </param>
2318            /// <param name="startAt">
2319            /// <para>The start at.</para>
2320            /// <para></para>
2321            /// </param>
2322            /// <returns>
2323            /// <para>A hash set of ulong</para>
2324            /// <para></para>
2325            /// </returns>
2326            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2327            private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
                ↪ HashSet<ulong> previousMatchings, long startAt)
2328            {
2329                if (startAt >= sequence.Length) // ?
2330                {
2331                    return previousMatchings;
2332                }
2333                var secondLinkUsages = new HashSet<ulong>();
2334                AllUsagesCore(sequence[startAt], secondLinkUsages);
2335                secondLinkUsages.Add(sequence[startAt]);
2336                var matchings = new HashSet<ulong>();
2337                var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
2338                //for (var i = 0; i < previousMatchings.Count; i++)
2339                foreach (var secondLinkUsage in secondLinkUsages)
2340                {
2341                    foreach (var previousMatching in previousMatchings)
2342                    {
2343                        //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                            ↪ secondLinkUsage);
2344                        StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                            ↪ secondLinkUsage);
2345                        TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                            ↪ previousMatching);
2346                        //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                            ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
                            ↪ желаемым результам.
2347                        PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                            ↪ secondLinkUsage);
2348                    }
2349                }
2350                if (matchings.Count == 0)
2351                {
2352                    return matchings;
2353                }
2354                return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
2355            }
2356
2357            /// <summary>
2358            /// <para>
2359            /// Ensures the each link is any or zero or many or exists using the specified links.
2360            /// </para>
2361            /// <para></para>
2362            /// </summary>
2363            /// <param name="links">
2364            /// <para>The links.</para>
2365            /// <para></para>
2366            /// </param>
2367            /// <param name="sequence">
2368            /// <para>The sequence.</para>
2369            /// <para></para>
2370            /// </param>
```

```
        /// <exception cref="ArgumentLinkDoesNotExistsException{ulong}">
        /// <para>patternSequence[{i}]</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
        ↪  links, params ulong[] sequence)
        {
            if (sequence == null)
            {
                return;
            }
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
                ↪  !links.Exists(sequence[i]))
                {
                    throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
                    ↪  $"patternSequence[{i}]");
                }
            }
        }

        // Pattern Matching -> Key To Triggers
        /// <summary>
        /// <para>
        /// Matches the pattern using the specified pattern sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="patternSequence">
        /// <para>The pattern sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                patternSequence = Simplify(patternSequence);
                if (patternSequence.Length > 0)
                {
                    EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                    var uniqueSequenceElements = new HashSet<ulong>();
                    for (var i = 0; i < patternSequence.Length; i++)
                    {
                        if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                        ↪  ZeroOrMany)
                        {
                            uniqueSequenceElements.Add(patternSequence[i]);
                        }
                    }
                    var results = new HashSet<ulong>();
                    foreach (var uniqueSequenceElement in uniqueSequenceElements)
                    {
                        AllUsagesCore(uniqueSequenceElement, results);
                    }
                    var filteredResults = new HashSet<ulong>();
                    var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                    matcher.AddAllPatternMatchedToResults(results);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        // Найти все возможные связи между указанным списком связей.
        // Находит связи между всеми указанными связями в любом порядке.
        // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
        ↪  несколько раз в последовательности)
        /// <summary>
        /// <para>
        /// Gets the all connections using the specified links to connect.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksToConnect">
        /// <para>The links to connect.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        AllUsagesCore(linksToConnect[i], next);
                        results.IntersectWith(next);
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all connections 1 using the specified links to connect.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksToConnect">
        /// <para>The links to connect.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var collector = new AllUsagesCollector(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results.IntersectWith(next);
                        next.Clear();
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all connections 2 using the specified links to connect.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksToConnect">
        /// <para>The links to connect.</para>
```

```
2521            /// <para></para>
2522            /// </param>
2523            /// <returns>
2524            /// <para>A hash set of ulong</para>
2525            /// <para></para>
2526            /// </returns>
2527            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2528            public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
2529            {
2530                return _sync.ExecuteReadOperation(() =>
2531                {
2532                    var results = new HashSet<ulong>();
2533                    if (linksToConnect.Length > 0)
2534                    {
2535                        Links.EnsureLinkExists(linksToConnect);
2536                        var collector1 = new AllUsagesCollector(Links, results);
2537                        collector1.Collect(linksToConnect[0]);
2538                        //AllUsagesCore(linksToConnect[0], results);
2539                        for (var i = 1; i < linksToConnect.Length; i++)
2540                        {
2541                            var next = new HashSet<ulong>();
2542                            var collector = new AllUsagesIntersectingCollector(Links, results, next);
2543                            collector.Collect(linksToConnect[i]);
2544                            //AllUsagesCore(linksToConnect[i], next);
2545                            //results.IntersectWith(next);
2546                            results = next;
2547                        }
2548                    }
2549                    return results;
2550                });
2551            }
2552
2553            /// <summary>
2554            /// <para>
2555            /// Gets the all connections 3 using the specified links to connect.
2556            /// </para>
2557            /// <para></para>
2558            /// </summary>
2559            /// <param name="linksToConnect">
2560            /// <para>The links to connect.</para>
2561            /// <para></para>
2562            /// </param>
2563            /// <returns>
2564            /// <para>A list of ulong</para>
2565            /// <para></para>
2566            /// </returns>
2567            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2568            public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
2569            {
2570                return _sync.ExecuteReadOperation(() =>
2571                {
2572                    var results = new BitString((long)Links.Unsync.Count() + 1); // new
                    ↪  BitArray((int)_links.Total + 1);
2573                    if (linksToConnect.Length > 0)
2574                    {
2575                        Links.EnsureLinkExists(linksToConnect);
2576                        var collector1 = new AllUsagesCollector2(Links.Unsync, results);
2577                        collector1.Collect(linksToConnect[0]);
2578                        for (var i = 1; i < linksToConnect.Length; i++)
2579                        {
2580                            var next = new BitString((long)Links.Unsync.Count() + 1); //new
                            ↪  BitArray((int)_links.Total + 1);
2581                            var collector = new AllUsagesCollector2(Links.Unsync, next);
2582                            collector.Collect(linksToConnect[i]);
2583                            results = results.And(next);
2584                        }
2585                    }
2586                    return results.GetSetUInt64Indices();
2587                });
2588            }
2589
2590            /// <summary>
2591            /// <para>
2592            /// Simplifies the sequence.
2593            /// </para>
2594            /// <para></para>
2595            /// </summary>
2596            /// <param name="sequence">
```

```
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The new sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newLength++;
            }
            // Строим новую последовательность
            zeroOrManyStepped = false;
            var newSequence = new ulong[newLength];
            long j = 0;
            for (var i = 0; i < sequence.Length; i++)
            {
                //var current = zeroOrManyStepped;
                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (current && zeroOrManyStepped)
                //    continue;
                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (zeroOrManyStepped && newZeroOrManyStepped)
                //    continue;
                //zeroOrManyStepped = newZeroOrManyStepped;
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newSequence[j++] = sequence[i];
            }
            return newSequence;
        }

        /// <summary>
        /// <para>
        /// Tests the simplify.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void TestSimplify()
        {
            var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
            ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
            var simplifiedSequence = Simplify(sequence);
        }

        /// <summary>
        /// <para>
        /// Gets the similar sequences.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSimilarSequences() => new List<ulong>();

        /// <summary>
        /// <para>
        /// Predictions this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Prediction()
        {
            //_links
            //sequences
        }

        #region From Triplets

        //public static void DeleteSequence(Link sequence)
        //{
        //}

        /// <summary>
        /// <para>
        /// Collects the matching sequences using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The results.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CollectMatchingSequences(ulong[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("Подпоследовательности с одним элементом не
                ↪  поддерживаются.");
            }
            var leftBound = 0;
            var rightBound = links.Length - 1;
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<ulong>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }

        /// <summary>
        /// <para>
        /// Collects the matching sequences using the specified left link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftBound">
        /// <para>The left bound.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="middleLinks">
        /// <para>The middle links.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightBound">
        /// <para>The right bound.</para>
        /// <para></para>
        /// </param>
        /// <param name="results">
        /// <para>The results.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪  middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
        {
            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
            if (leftLinkTotalReferers <= rightLinkTotalReferers)
            {
                var nextLeftLink = middleLinks[leftBound];
                var elements = GetRightElements(leftLink, nextLeftLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                            ↪  rightLink, rightBound, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
            else
            {
                var nextRightLink = middleLinks[rightBound];
                var elements = GetLeftElements(rightLink, nextRightLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                            ↪  elements[i], rightBound - 1, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != 0)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
```

```
        }

        /// <summary>
        /// <para>
        /// Gets the right elements using the specified start link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
        {
            var result = new ulong[5];
            TryStepRight(startLink, rightLink, result, 0);
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepRight(couple, rightLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
            {
                result[4] = startLink;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance try step right.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    var coupleTarget = Links.GetTarget(couple);
```

```
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        /// <summary>
        /// <para>
        /// Gets the left elements using the specified start link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
        {
            var result = new ulong[5];
            TryStepLeft(startLink, leftLink, result, 0);
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepLeft(couple, leftLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
            {
                result[4] = leftLink;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance try step left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    var coupleSource = Links.GetSource(couple);
                    if (coupleSource == leftLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                     ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        #endregion

        #region Walkers

        /// <summary>
        /// <para>
        /// Represents the pattern matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="RightSequenceWalker{ulong}"/>
        public class PatternMatcher : RightSequenceWalker<ulong>
        {
            /// <summary>
            /// <para>
            /// The sequences.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly Sequences _sequences;
            /// <summary>
            /// <para>
            /// The pattern sequence.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly ulong[] _patternSequence;
            /// <summary>
            /// <para>
            /// The links in sequence.
            /// </para>
            /// <para></para>
            /// </summary>
            private readonly HashSet<LinkIndex> _linksInSequence;
            /// <summary>
```

```
        /// <para>
        /// The results.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly HashSet<LinkIndex> _results;

        #region Pattern Match

        /// <summary>
        /// <para>
        /// The pattern block type enum.
        /// </para>
        /// <para></para>
        /// </summary>
        enum PatternBlockType
        {
            /// <summary>
            /// <para>
            /// The undefined pattern block type.
            /// </para>
            /// <para></para>
            /// </summary>
            Undefined,
            /// <summary>
            /// <para>
            /// The gap pattern block type.
            /// </para>
            /// <para></para>
            /// </summary>
            Gap,
            /// <summary>
            /// <para>
            /// The elements pattern block type.
            /// </para>
            /// <para></para>
            /// </summary>
            Elements
        }

        /// <summary>
        /// <para>
        /// The pattern block.
        /// </para>
        /// <para></para>
        /// </summary>
        struct PatternBlock
        {
            /// <summary>
            /// <para>
            /// The type.
            /// </para>
            /// <para></para>
            /// </summary>
            public PatternBlockType Type;
            /// <summary>
            /// <para>
            /// The start.
            /// </para>
            /// <para></para>
            /// </summary>
            public long Start;
            /// <summary>
            /// <para>
            /// The stop.
            /// </para>
            /// <para></para>
            /// </summary>
            public long Stop;
        }

        /// <summary>
        /// <para>
        /// The pattern.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly List<PatternBlock> _pattern;
        /// <summary>
```

```
3139            /// <para>
3140            /// The pattern position.
3141            /// </para>
3142            /// <para></para>
3143            /// </summary>
3144            private int _patternPosition;
3145            /// <summary>
3146            /// <para>
3147            /// The sequence position.
3148            /// </para>
3149            /// <para></para>
3150            /// </summary>
3151            private long _sequencePosition;
3152
3153            #endregion
3154
3155            /// <summary>
3156            /// <para>
3157            /// Initializes a new <see cref="PatternMatcher"/> instance.
3158            /// </para>
3159            /// <para></para>
3160            /// </summary>
3161            /// <param name="sequences">
3162            /// <para>A sequences.</para>
3163            /// <para></para>
3164            /// </param>
3165            /// <param name="patternSequence">
3166            /// <para>A pattern sequence.</para>
3167            /// <para></para>
3168            /// </param>
3169            /// <param name="results">
3170            /// <para>A results.</para>
3171            /// <para></para>
3172            /// </param>
3173            [MethodImpl(MethodImplOptions.AggressiveInlining)]
3174            public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                ↪  HashSet<LinkIndex> results)
3175                : base(sequences.Links.Unsync, new DefaultStack<ulong>())
3176            {
3177                _sequences = sequences;
3178                _patternSequence = patternSequence;
3179                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _sequences.Constants.Any && x != ZeroOrMany));
3180                _results = results;
3181                _pattern = CreateDetailedPattern();
3182            }
3183
3184            /// <summary>
3185            /// <para>
3186            /// Determines whether this instance is element.
3187            /// </para>
3188            /// <para></para>
3189            /// </summary>
3190            /// <param name="link">
3191            /// <para>The link.</para>
3192            /// <para></para>
3193            /// </param>
3194            /// <returns>
3195            /// <para>The bool</para>
3196            /// <para></para>
3197            /// </returns>
3198            [MethodImpl(MethodImplOptions.AggressiveInlining)]
3199            protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
                ↪  base.IsElement(link);
3200
3201            /// <summary>
3202            /// <para>
3203            /// Determines whether this instance pattern match.
3204            /// </para>
3205            /// <para></para>
3206            /// </summary>
3207            /// <param name="sequenceToMatch">
3208            /// <para>The sequence to match.</para>
3209            /// <para></para>
3210            /// </param>
3211            /// <returns>
3212            /// <para>The bool</para>
3213            /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool PatternMatch(LinkIndex sequenceToMatch)
        {
            _patternPosition = 0;
            _sequencePosition = 0;
            foreach (var part in Walk(sequenceToMatch))
            {
                if (!PatternMatchCore(part))
                {
                    break;
                }
            }
            return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
            ↪  - 1 && _pattern[_patternPosition].Start == 0);
        }

        /// <summary>
        /// <para>
        /// Creates the detailed pattern.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The pattern.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<PatternBlock> CreateDetailedPattern()
        {
            var pattern = new List<PatternBlock>();
            var patternBlock = new PatternBlock();
            for (var i = 0; i < _patternSequence.Length; i++)
            {
                if (patternBlock.Type == PatternBlockType.Undefined)
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        patternBlock.Type = PatternBlockType.Gap;
                        patternBlock.Start = 1;
                        patternBlock.Stop = 1;
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Type = PatternBlockType.Gap;
                        patternBlock.Start = 0;
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        patternBlock.Type = PatternBlockType.Elements;
                        patternBlock.Start = i;
                        patternBlock.Stop = i;
                    }
                }
                else if (patternBlock.Type == PatternBlockType.Elements)
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 1,
                            Stop = 1
                        };
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 0,
                            Stop = long.MaxValue
                        };
                    }
                    else
                    {
                        patternBlock.Stop = i;
```

```
                    }
                }
                else // patternBlock.Type == PatternBlockType.Gap
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        patternBlock.Start++;
                        if (patternBlock.Stop < patternBlock.Start)
                        {
                            patternBlock.Stop = patternBlock.Start;
                        }
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Elements,
                            Start = i,
                            Stop = i
                        };
                    }
                }
            }
            if (patternBlock.Type != PatternBlockType.Undefined)
            {
                pattern.Add(patternBlock);
            }
            return pattern;
        }

        // match: search for regexp anywhere in text
        //int match(char* regexp, char* text)
        //{
        //    do
        //    {
        //    } while (*text++ != '\0');
        //    return 0;
        //}

        // matchhere: search for regexp at beginning of text
        //int matchhere(char* regexp, char* text)
        //{
        //    if (regexp[0] == '\0')
        //        return 1;
        //    if (regexp[1] == '*')
        //        return matchstar(regexp[0], regexp + 2, text);
        //    if (regexp[0] == '$' && regexp[1] == '\0')
        //        return *text == '\0';
        //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        //        return matchhere(regexp + 1, text + 1);
        //    return 0;
        //}

        // matchstar: search for c*regexp at beginning of text
        //int matchstar(int c, char* regexp, char* text)
        //{
        //    do
        //    {    /* a * matches zero or more instances */
        //        if (matchhere(regexp, text))
        //            return 1;
        //    } while (*text != '\0' && (*text++ == c || c == '.'));
        //    return 0;
        //}

        //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
        ↪  long maximumGap)
        //{
        //    mininumGap = 0;
        //    maximumGap = 0;
        //    element = 0;
        //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
        //    {
        //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
        //            mininumGap++;
```

```
3371                //         else if (_patternSequence[_patternPosition] == ZeroOrMany)
3372                //             maximumGap = long.MaxValue;
3373                //         else
3374                //             break;
3375                //     }
3376
3377                //     if (maximumGap < mininumGap)
3378                //         maximumGap = mininumGap;
3379                //}
3380
3381                /// <summary>
3382                /// <para>
3383                /// Determines whether this instance pattern match core.
3384                /// </para>
3385                /// <para></para>
3386                /// </summary>
3387                /// <param name="element">
3388                /// <para>The element.</para>
3389                /// <para></para>
3390                /// </param>
3391                /// <returns>
3392                /// <para>The bool</para>
3393                /// <para></para>
3394                /// </returns>
3395                [MethodImpl(MethodImplOptions.AggressiveInlining)]
3396                private bool PatternMatchCore(LinkIndex element)
3397                {
3398                    if (_patternPosition >= _pattern.Count)
3399                    {
3400                        _patternPosition = -2;
3401                        return false;
3402                    }
3403                    var currentPatternBlock = _pattern[_patternPosition];
3404                    if (currentPatternBlock.Type == PatternBlockType.Gap)
3405                    {
3406                        //var currentMatchingBlockLength = (_sequencePosition -
                            ↪ _lastMatchedBlockPosition);
3407                        if (_sequencePosition < currentPatternBlock.Start)
3408                        {
3409                            _sequencePosition++;
3410                            return true; // Двигаемся дальше
3411                        }
3412                        // Это последний блок
3413                        if (_pattern.Count == _patternPosition + 1)
3414                        {
3415                            _patternPosition++;
3416                            _sequencePosition = 0;
3417                            return false; // Полное соответствие
3418                        }
3419                        else
3420                        {
3421                            if (_sequencePosition > currentPatternBlock.Stop)
3422                            {
3423                                return false; // Соответствие невозможно
3424                            }
3425                            var nextPatternBlock = _pattern[_patternPosition + 1];
3426                            if (_patternSequence[nextPatternBlock.Start] == element)
3427                            {
3428                                if (nextPatternBlock.Start < nextPatternBlock.Stop)
3429                                {
3430                                    _patternPosition++;
3431                                    _sequencePosition = 1;
3432                                }
3433                                else
3434                                {
3435                                    _patternPosition += 2;
3436                                    _sequencePosition = 0;
3437                                }
3438                            }
3439                        }
3440                    }
3441                    else // currentPatternBlock.Type == PatternBlockType.Elements
3442                    {
3443                        var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
3444                        if (_patternSequence[patternElementPosition] != element)
3445                        {
3446                            return false; // Соответствие невозможно
3447                        }
3448                        if (patternElementPosition == currentPatternBlock.Stop)
```

```
                    {
                        _patternPosition++;
                        _sequencePosition = 0;
                    }
                    else
                    {
                        _sequencePosition++;
                    }
                }
                return true;
                //if (_patternSequence[_patternPosition] != element)
                //    return false;
                //else
                //{
                //    _sequencePosition++;
                //    _patternPosition++;
                //    return true;
                //}
                ////////
                //if (_filterPosition == _patternSequence.Length)
                //{
                //    _filterPosition = -2; // Длиннее чем нужно
                //    return false;
                //}
                //if (element != _patternSequence[_filterPosition])
                //{
                //    _filterPosition = -1;
                //    return false; // Начинается иначе
                //}
                //_filterPosition++;
                //if (_filterPosition == (_patternSequence.Length - 1))
                //    return false;
                //if (_filterPosition >= 0)
                //{
                //    if (element == _patternSequence[_filterPosition + 1])
                //        _filterPosition++;
                //    else
                //        return false;
                //}
                //if (_filterPosition < 0)
                //{
                //    if (element == _patternSequence[0])
                //        _filterPosition = 0;
                //}
            }

            /// <summary>
            /// <para>
            /// Adds the all pattern matched to results using the specified sequences to match.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="sequencesToMatch">
            /// <para>The sequences to match.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PatternMatch(sequenceToMatch))
                    {
                        _results.Add(sequenceToMatch);
                    }
                }
            }

            #endregion
        }
    }
```

## 1.43 ./csharp/Platform.Data.Doublets.Sequences/Sequences.cs

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

```
using Platform.Collections.Lists;
using Platform.Collections.Stacks;
using Platform.Threading.Synchronization;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// Представляет коллекцию последовательностей связей.
    /// </summary>
    /// <remarks>
    /// Обязательно реализовать атомарность каждого публичного метода.
    ///
    /// TODO:
    ///
    /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
    /// через естественную группировку по unicode типам, все whitespace вместе, все символы вместе, все числа вместе и т.п.
    /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
    ///
    /// x*y - найти все связи между, в последовательностях любой формы, если не стоит ограничитель на то, что является последовательностью, а что нет,
    /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
    ///
    /// Рост последовательности слева и справа.
    /// Поиск со звёздочкой.
    /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
    /// так же проблема может быть решена при реализации дистанционных триггеров.
    /// Нужны ли уникальные указатели вообще?
    /// Что если обращение к информации будет происходить через содержимое всегда?
    ///
    /// Писать тесты.
    ///
    ///
    /// Можно убрать зависимость от конкретной реализации Links,
    /// на зависимость от абстрактного элемента, который может быть представлен несколькими способами.
    ///
    /// Можно ли как-то сделать один общий интерфейс
    ///
    ///
    /// Блокчейн и/или гит для распределённой записи транзакций.
    ///
    /// </remarks>
    public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]> (после завершения реализации Sequences)
    {
        /// <summary>Возвращает значение LinkIndex, обозначающее любое количество связей.</summary>
        public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;

        /// <summary>
        /// <para>
        /// Gets the options value.
        /// </para>
        /// <para></para>
        /// </summary>
        public SequencesOptions<LinkIndex> Options { get; }
        /// <summary>
        /// <para>
        /// Gets the links value.
        /// </para>
        /// <para></para>
        /// </summary>
        public SynchronizedLinks<LinkIndex> Links { get; }
        /// <summary>
        /// <para>
        /// The sync.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ISynchronization _sync;

```

```
        /// <summary>
        /// <para>
        /// Gets the constants value.
        /// </para>
        /// <para></para>
        /// </summary>
        public LinksConstants<LinkIndex> Constants { get; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Sequences"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="options">
        /// <para>A options.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
        {
            Links = links;
            _sync = links.SyncRoot;
            Options = options;
            Options.ValidateOptions();
            Options.InitOptions(Links);
            Constants = links.Constants;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Sequences"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
        ↪  SequencesOptions<LinkIndex>()) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance is sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsSequence(LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }

        /// <summary>
        /// <para>
        /// Gets the sequence by elements using the specified sequence.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceByElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
            }
            return sequence;
        }

        /// <summary>
        /// <para>
        /// Gets the sequence elements using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                var linkContents = new Link<ulong>(Links.GetLink(sequence));
                if (linkContents.Source == Options.SequenceMarkerLink)
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        /// <summary>
        /// <para>
        /// Counts the restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Count(IList<LinkIndex> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
```

```
            }
            if (restrictions.Count == 1) // Первая связь это адрес
            {
                var sequenceIndex = restrictions[0];
                if (sequenceIndex == Constants.Null)
                {
                    return 0;
                }
                if (sequenceIndex == Constants.Any)
                {
                    return Count(null);
                }
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
                }
                return Links.Exists(sequenceIndex) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }

        /// <summary>
        /// <para>
        /// Counts the usages using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CountUsages(params LinkIndex[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == Constants.Null)
                {
                    return 0;
                }
                var any = Constants.Any;
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != Constants.Null)
                    {
                        return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                        ↪  1;
                    }
                    return Links.Count(any, elementsLink);
                }
                return Links.Count(any, restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

        /// <summary>
        /// <para>
        /// Creates the restrictions.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Create(IList<LinkIndex> restrictions)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (restrictions.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
                return CreateCore(restrictions);
            });
        }

        /// <summary>
        /// <para>
        /// Creates the core using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The sequence root.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CreateCore(IList<LinkIndex> restrictions)
        {
            LinkIndex[] sequence = restrictions.SkipFirst();
            if (Options.UseIndex)
            {
                Options.Index.Add(sequence);
            }
            var sequenceRoot = default(LinkIndex);
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
            {
                var matches = Each(restrictions);
                if (matches.Count > 0)
                {
                    sequenceRoot = matches[0];
                }
            }
            else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
            {
                return CompactCore(sequence);
            }
            if (sequenceRoot == default)
            {
                sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
            }
            if (Options.UseSequenceMarker)
            {
                return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
            }
            return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
        }

        #endregion

        #region Each

        /// <summary>
        /// <para>
        /// Eaches the sequence.
        /// </para>
        /// <para></para>
```

```
387             /// </summary>
388             /// <param name="sequence">
389             /// <para>The sequence.</para>
390             /// <para></para>
391             /// </param>
392             /// <returns>
393             /// <para>The results.</para>
394             /// <para></para>
395             /// </returns>
396             [MethodImpl(MethodImplOptions.AggressiveInlining)]
397             public List<LinkIndex> Each(IList<LinkIndex> sequence)
398             {
399                 var results = new List<LinkIndex>();
400                 var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
401                 Each(filler.AddFirstAndReturnConstant, sequence);
402                 return results;
403             }
404
405             /// <summary>
406             /// <para>
407             /// Eaches the handler.
408             /// </para>
409             /// <para></para>
410             /// </summary>
411             /// <param name="handler">
412             /// <para>The handler.</para>
413             /// <para></para>
414             /// </param>
415             /// <param name="restrictions">
416             /// <para>The restrictions.</para>
417             /// <para></para>
418             /// </param>
419             /// <exception cref="NotImplementedException">
420             /// <para></para>
421             /// <para></para>
422             /// </exception>
423             /// <returns>
424             /// <para>The link index</para>
425             /// <para></para>
426             /// </returns>
427             [MethodImpl(MethodImplOptions.AggressiveInlining)]
428             public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
                ↪ restrictions)
429             {
430                 return _sync.ExecuteReadOperation(() =>
431                 {
432                     if (restrictions.IsNullOrEmpty())
433                     {
434                         return Constants.Continue;
435                     }
436                     Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
437                     if (restrictions.Count == 1)
438                     {
439                         var link = restrictions[0];
440                         var any = Constants.Any;
441                         if (link == any)
442                         {
443                             if (Options.UseSequenceMarker)
444                             {
445                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                    ↪ Options.SequenceMarkerLink, any));
446                             }
447                             else
448                             {
449                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                    ↪ any));
450                             }
451                         }
452                         if (Options.UseSequenceMarker)
453                         {
454                             var sequenceLinkValues = Links.Unsync.GetLink(link);
455                             if (sequenceLinkValues[Constants.SourcePart] ==
                                ↪ Options.SequenceMarkerLink)
456                             {
457                                 link = sequenceLinkValues[Constants.TargetPart];
458                             }
459                         }
460                         var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
```

```
                    sequence[0] = link;
                    return handler(sequence);
                }
                else if (restrictions.Count == 2)
                {
                    throw new NotImplementedException();
                }
                else if (restrictions.Count == 3)
                {
                    return Links.Unsync.Each(handler, restrictions);
                }
                else
                {
                    var sequence = restrictions.SkipFirst();
                    if (Options.UseIndex && !Options.Index.MightContain(sequence))
                    {
                        return Constants.Break;
                    }
                    return EachCore(handler, sequence);
                }
            });
        }

        /// <summary>
        /// <para>
        /// Eaches the core using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="values">
        /// <para>The values.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪  values)
        {
            var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
            // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            ↪  Id.
            Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
            ↪  (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
            ↪  matcher.HandleFullMatched;
            //if (sequence.Length >= 2)
            if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
            {
                return Constants.Break;
            }
            var last = values.Count - 2;
            for (var i = 1; i < last; i++)
            {
                if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                ↪  Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            if (values.Count >= 3)
            {
                if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                ↪  != Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            return Constants.Continue;
        }

        /// <summary>
        /// <para>
```

```
533             /// Partials the step right using the specified handler.
534             /// </para>
535             /// <para></para>
536             /// </summary>
537             /// <param name="handler">
538             /// <para>The handler.</para>
539             /// <para></para>
540             /// </param>
541             /// <param name="left">
542             /// <para>The left.</para>
543             /// <para></para>
544             /// </param>
545             /// <param name="right">
546             /// <para>The right.</para>
547             /// <para></para>
548             /// </param>
549             /// <returns>
550             /// <para>The link index</para>
551             /// <para></para>
552             /// </returns>
553             [MethodImpl(MethodImplOptions.AggressiveInlining)]
554             private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
                ↪  left, LinkIndex right)
555             {
556                 return Links.Unsync.Each(doublet =>
557                 {
558                     var doubletIndex = doublet[Constants.IndexPart];
559                     if (StepRight(handler, doubletIndex, right) != Constants.Continue)
560                     {
561                         return Constants.Break;
562                     }
563                     if (left != doubletIndex)
564                     {
565                         return PartialStepRight(handler, doubletIndex, right);
566                     }
567                     return Constants.Continue;
568                 }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
569             }
570
571             /// <summary>
572             /// <para>
573             /// Steps the right using the specified handler.
574             /// </para>
575             /// <para></para>
576             /// </summary>
577             /// <param name="handler">
578             /// <para>The handler.</para>
579             /// <para></para>
580             /// </param>
581             /// <param name="left">
582             /// <para>The left.</para>
583             /// <para></para>
584             /// </param>
585             /// <param name="right">
586             /// <para>The right.</para>
587             /// <para></para>
588             /// </param>
589             /// <returns>
590             /// <para>The link index</para>
591             /// <para></para>
592             /// </returns>
593             [MethodImpl(MethodImplOptions.AggressiveInlining)]
594             private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
                ↪  LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
                ↪  rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
                ↪  Constants.Any));
595
596             /// <summary>
597             /// <para>
598             /// Tries the step right up using the specified handler.
599             /// </para>
600             /// <para></para>
601             /// </summary>
602             /// <param name="handler">
603             /// <para>The handler.</para>
604             /// <para></para>
605             /// </param>
606             /// <param name="right">
```

```
607             /// <para>The right.</para>
608             /// <para></para>
609             /// </param>
610             /// <param name="stepFrom">
611             /// <para>The step from.</para>
612             /// <para></para>
613             /// </param>
614             /// <returns>
615             /// <para>The link index</para>
616             /// <para></para>
617             /// </returns>
618             [MethodImpl(MethodImplOptions.AggressiveInlining)]
619             private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
                ↪ right, LinkIndex stepFrom)
620             {
621                 var upStep = stepFrom;
622                 var firstSource = Links.Unsync.GetTarget(upStep);
623                 while (firstSource != right && firstSource != upStep)
624                 {
625                     upStep = firstSource;
626                     firstSource = Links.Unsync.GetSource(upStep);
627                 }
628                 if (firstSource == right)
629                 {
630                     return handler(new LinkAddress<LinkIndex>(stepFrom));
631                 }
632                 return Constants.Continue;
633             }
634
635             /// <summary>
636             /// <para>
637             /// Steps the left using the specified handler.
638             /// </para>
639             /// <para></para>
640             /// </summary>
641             /// <param name="handler">
642             /// <para>The handler.</para>
643             /// <para></para>
644             /// </param>
645             /// <param name="left">
646             /// <para>The left.</para>
647             /// <para></para>
648             /// </param>
649             /// <param name="right">
650             /// <para>The right.</para>
651             /// <para></para>
652             /// </param>
653             /// <returns>
654             /// <para>The link index</para>
655             /// <para></para>
656             /// </returns>
657             [MethodImpl(MethodImplOptions.AggressiveInlining)]
658             private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
                ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
                ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
                ↪ right));
659
660             /// <summary>
661             /// <para>
662             /// Tries the step left up using the specified handler.
663             /// </para>
664             /// <para></para>
665             /// </summary>
666             /// <param name="handler">
667             /// <para>The handler.</para>
668             /// <para></para>
669             /// </param>
670             /// <param name="left">
671             /// <para>The left.</para>
672             /// <para></para>
673             /// </param>
674             /// <param name="stepFrom">
675             /// <para>The step from.</para>
676             /// <para></para>
677             /// </param>
678             /// <returns>
679             /// <para>The link index</para>
680             /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪  left, LinkIndex stepFrom)
        {
            var upStep = stepFrom;
            var firstTarget = Links.Unsync.GetSource(upStep);
            while (firstTarget != left && firstTarget != upStep)
            {
                upStep = firstTarget;
                firstTarget = Links.Unsync.GetTarget(upStep);
            }
            if (firstTarget == left)
            {
                return handler(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        #endregion

        #region Update

        /// <summary>
        /// <para>
        /// Updates the restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
        {
            var sequence = restrictions.SkipFirst();
            var newSequence = substitution.SkipFirst();
            if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
            {
                return Constants.Null;
            }
            if (sequence.IsNullOrEmpty())
            {
                return Create(substitution);
            }
            if (newSequence.IsNullOrEmpty())
            {
                Delete(restrictions);
                return Constants.Null;
            }
            return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
            {
                ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
                Links.EnsureLinkExists(newSequence);
                return UpdateCore(sequence, newSequence);
            }));
        }

        /// <summary>
        /// <para>
        /// Updates the core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSequence">
```

```
        /// <para>The new sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The best variant.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
        {
            LinkIndex bestVariant;
            if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
            ↪  !sequence.EqualTo(newSequence))
            {
                bestVariant = CompactCore(newSequence);
            }
            else
            {
                bestVariant = CreateCore(newSequence);
            }
            // TODO: Check all options only ones before loop execution
            // Возможно нужно две версии Each, возвращающий фактические последовательности и с
            ↪  маркером,
            // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
            ↪  можно получить имея только фактические последовательности.
            foreach (var variant in Each(sequence))
            {
                if (variant != bestVariant)
                {
                    UpdateOneCore(variant, bestVariant);
                }
            }
            return bestVariant;
        }

        /// <summary>
        /// <para>
        /// Updates the one core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSequence">
        /// <para>The new sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(sequence);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                var newSequenceElements = GetSequenceElements(newSequence);
                var newSequenceLink = GetSequenceByElements(newSequenceElements);
                if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                    }
                    Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(sequence);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    var newSequenceElements = GetSequenceElements(newSequence);
                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
```

```
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
                        }
                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
                    }
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeAndDelete(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

        /// <summary>
        /// <para>
        /// Deletes the restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Delete(IList<LinkIndex> restrictions)
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequence = restrictions.SkipFirst();
                // TODO: Check all options only ones before loop execution
                foreach (var linkToDelete in Each(sequence))
                {
                    DeleteOneCore(linkToDelete);
                }
            });
        }

        /// <summary>
        /// <para>
        /// Deletes the one core using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void DeleteOneCore(LinkIndex link)
        {
            if (Options.UseGarbageCollection)
            {
                var sequenceElements = GetSequenceElements(link);
                var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
                var sequenceLink = GetSequenceByElements(sequenceElements);
                if (Options.UseCascadeDelete || CountUsages(link) == 0)
                {
                    if (sequenceLink != Constants.Null)
                    {
                        Links.Unsync.Delete(sequenceLink);
                    }
                    Links.Unsync.Delete(link);
                }
                ClearGarbage(sequenceElementsContents.Source);
                ClearGarbage(sequenceElementsContents.Target);
            }
            else
            {
```

```
                if (Options.UseSequenceMarker)
                {
                    var sequenceElements = GetSequenceElements(link);
                    var sequenceLink = GetSequenceByElements(sequenceElements);
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        if (sequenceLink != Constants.Null)
                        {
                            Links.Unsync.Delete(sequenceLink);
                        }
                        Links.Unsync.Delete(link);
                    }
                }
                else
                {
                    if (Options.UseCascadeDelete || CountUsages(link) == 0)
                    {
                        Links.Unsync.Delete(link);
                    }
                }
            }
        }

        #endregion

        #region Compactification

        /// <summary>
        /// <para>
        /// Compacts the all.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CompactAll()
        {
            _sync.ExecuteWriteOperation(() =>
            {
                var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
                for (int i = 0; i < sequences.Count; i++)
                {
                    var sequence = this.ToList(sequences[i]);
                    Compact(sequence.ShiftRight());
                }
            });
        }

        /// <remarks>
        /// bestVariant можно выбирать по максимальному числу использований,
        /// но балансированный позволяет гарантировать уникальность (если есть возможность,
        /// гарантировать его использование в других местах).
        ///
        /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Compact(IList<LinkIndex> sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Constants.Null;
                }
                Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
                return CompactCore(sequence);
            });
        }

        /// <summary>
        /// <para>
        /// Compacts the core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
        ↪  sequence);

        #endregion

        #region Garbage Collection

        /// <remarks>
        /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
        ↪  определить извне или в унаследованном классе
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
        ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;

        /// <summary>
        /// <para>
        /// Clears the garbage using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ClearGarbage(LinkIndex link)
        {
            if (IsGarbage(link))
            {
                var contents = new Link<ulong>(Links.GetLink(link));
                Links.Unsync.Delete(link);
                ClearGarbage(contents.Source);
                ClearGarbage(contents.Target);
            }
        }

        #endregion

        #region Walkers

        /// <summary>
        /// <para>
        /// Determines whether this instance each part.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var links = Links.Unsync;
                foreach (var part in Options.Walker.Walk(sequence))
                {
                    if (!handler(part))
                    {
                        return false;
                    }
                }
                return true;
            });
```

```
1065            }
1066
1067            /// <summary>
1068            /// <para>
1069            /// Represents the matcher.
1070            /// </para>
1071            /// <para></para>
1072            /// </summary>
1073            /// <seealso cref="RightSequenceWalker{LinkIndex}"/>
1074            public class Matcher : RightSequenceWalker<LinkIndex>
1075            {
1076                /// <summary>
1077                /// <para>
1078                /// The sequences.
1079                /// </para>
1080                /// <para></para>
1081                /// </summary>
1082                private readonly Sequences _sequences;
1083                /// <summary>
1084                /// <para>
1085                /// The pattern sequence.
1086                /// </para>
1087                /// <para></para>
1088                /// </summary>
1089                private readonly IList<LinkIndex> _patternSequence;
1090                /// <summary>
1091                /// <para>
1092                /// The links in sequence.
1093                /// </para>
1094                /// <para></para>
1095                /// </summary>
1096                private readonly HashSet<LinkIndex> _linksInSequence;
1097                /// <summary>
1098                /// <para>
1099                /// The results.
1100                /// </para>
1101                /// <para></para>
1102                /// </summary>
1103                private readonly HashSet<LinkIndex> _results;
1104                /// <summary>
1105                /// <para>
1106                /// The stopable handler.
1107                /// </para>
1108                /// <para></para>
1109                /// </summary>
1110                private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
1111                /// <summary>
1112                /// <para>
1113                /// The read as elements.
1114                /// </para>
1115                /// <para></para>
1116                /// </summary>
1117                private readonly HashSet<LinkIndex> _readAsElements;
1118                /// <summary>
1119                /// <para>
1120                /// The filter position.
1121                /// </para>
1122                /// <para></para>
1123                /// </summary>
1124                private int _filterPosition;
1125
1126                /// <summary>
1127                /// <para>
1128                /// Initializes a new <see cref="Matcher"/> instance.
1129                /// </para>
1130                /// <para></para>
1131                /// </summary>
1132                /// <param name="sequences">
1133                /// <para>A sequences.</para>
1134                /// <para></para>
1135                /// </param>
1136                /// <param name="patternSequence">
1137                /// <para>A pattern sequence.</para>
1138                /// <para></para>
1139                /// </param>
1140                /// <param name="results">
1141                /// <para>A results.</para>
1142                /// <para></para>
```

```
1143            /// </param>
1144            /// <param name="stopableHandler">
1145            /// <para>A stopable handler.</para>
1146            /// <para></para>
1147            /// </param>
1148            /// <param name="readAsElements">
1149            /// <para>A read as elements.</para>
1150            /// <para></para>
1151            /// </param>
1152            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1153            public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
                ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
                ↪ HashSet<LinkIndex> readAsElements = null)
1154                : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
1155            {
1156                _sequences = sequences;
1157                _patternSequence = patternSequence;
1158                _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪ _links.Constants.Any && x != ZeroOrMany));
1159                _results = results;
1160                _stopableHandler = stopableHandler;
1161                _readAsElements = readAsElements;
1162            }
1163
1164            /// <summary>
1165            /// <para>
1166            /// Determines whether this instance is element.
1167            /// </para>
1168            /// <para></para>
1169            /// </summary>
1170            /// <param name="link">
1171            /// <para>The link.</para>
1172            /// <para></para>
1173            /// </param>
1174            /// <returns>
1175            /// <para>The bool</para>
1176            /// <para></para>
1177            /// </returns>
1178            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1179            protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪ _linksInSequence.Contains(link);
1180
1181            /// <summary>
1182            /// <para>
1183            /// Determines whether this instance full match.
1184            /// </para>
1185            /// <para></para>
1186            /// </summary>
1187            /// <param name="sequenceToMatch">
1188            /// <para>The sequence to match.</para>
1189            /// <para></para>
1190            /// </param>
1191            /// <returns>
1192            /// <para>The bool</para>
1193            /// <para></para>
1194            /// </returns>
1195            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1196            public bool FullMatch(LinkIndex sequenceToMatch)
1197            {
1198                _filterPosition = 0;
1199                foreach (var part in Walk(sequenceToMatch))
1200                {
1201                    if (!FullMatchCore(part))
1202                    {
1203                        break;
1204                    }
1205                }
1206                return _filterPosition == _patternSequence.Count;
1207            }
1208
1209            /// <summary>
1210            /// <para>
1211            /// Determines whether this instance full match core.
1212            /// </para>
1213            /// <para></para>
1214            /// </summary>
1215            /// <param name="element">
```

```
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool FullMatchCore(LinkIndex element)
        {
            if (_filterPosition == _patternSequence.Count)
            {
                _filterPosition = -2; // Длиннее чем нужно
                return false;
            }
            if (_patternSequence[_filterPosition] != _links.Constants.Any
             && element != _patternSequence[_filterPosition])
            {
                _filterPosition = -1;
                return false; // Начинается/Продолжается иначе
            }
            _filterPosition++;
            return true;
        }

        /// <summary>
        /// <para>
        /// Adds the full matched to results using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[_links.Constants.IndexPart];
            if (FullMatch(sequenceToMatch))
            {
                _results.Add(sequenceToMatch);
            }
        }

        /// <summary>
        /// <para>
        /// Handles the full matched using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[_links.Constants.IndexPart];
            if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
            {
                return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
            }
            return _links.Constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Handles the full matched sequence using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
```

```
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The link index</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
            {
                var sequenceToMatch = restrictions[_links.Constants.IndexPart];
                var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
                if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
                ↪  _results.Add(sequenceToMatch))
                {
                    return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
                }
                return _links.Constants.Continue;
            }

            /// <remarks>
            /// TODO: Add support for LinksConstants.Any
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool PartialMatch(LinkIndex sequenceToMatch)
            {
                _filterPosition = -1;
                foreach (var part in Walk(sequenceToMatch))
                {
                    if (!PartialMatchCore(part))
                    {
                        break;
                    }
                }
                return _filterPosition == _patternSequence.Count - 1;
            }

            /// <summary>
            /// <para>
            /// Determines whether this instance partial match core.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="element">
            /// <para>The element.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool PartialMatchCore(LinkIndex element)
            {
                if (_filterPosition == (_patternSequence.Count - 1))
                {
                    return false; // Нашлось
                }
                if (_filterPosition >= 0)
                {
                    if (element == _patternSequence[_filterPosition + 1])
                    {
                        _filterPosition++;
                    }
                    else
                    {
                        _filterPosition = -1;
                    }
                }
                if (_filterPosition < 0)
                {
                    if (element == _patternSequence[0])
                    {
                        _filterPosition = 0;
                    }
                }
                return true; // Ищем дальше
            }

```

```
        /// <summary>
        /// <para>
        /// Adds the partial matched to results using the specified sequence to match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceToMatch">
        /// <para>The sequence to match.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
        {
            if (PartialMatch(sequenceToMatch))
            {
                _results.Add(sequenceToMatch);
            }
        }

        /// <summary>
        /// <para>
        /// Handles the partial matched using the specified restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex HandlePartialMatched(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[_links.Constants.IndexPart];
            if (PartialMatch(sequenceToMatch))
            {
                return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
            }
            return _links.Constants.Continue;
        }

        /// <summary>
        /// <para>
        /// Adds the all partial matched to results using the specified sequences to match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequencesToMatch">
        /// <para>The sequences to match.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
        {
            foreach (var sequenceToMatch in sequencesToMatch)
            {
                if (PartialMatch(sequenceToMatch))
                {
                    _results.Add(sequenceToMatch);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Adds the all partial matched to results and read as elements using the specified
        ↪  sequences to match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequencesToMatch">
        /// <para>The sequences to match.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
1448            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
                ↪ sequencesToMatch)
1449            {
1450                foreach (var sequenceToMatch in sequencesToMatch)
1451                {
1452                    if (PartialMatch(sequenceToMatch))
1453                    {
1454                        _readAsElements.Add(sequenceToMatch);
1455                        _results.Add(sequenceToMatch);
1456                    }
1457                }
1458            }
1459        }
1460
1461        #endregion
1462    }
1463  }
```

## 1.44 ./csharp/Platform.Data.Doublets.Sequences/SequencesExtensions.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences
8   {
9       /// <summary>
10      /// <para>
11      /// Represents the sequences extensions.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      public static class SequencesExtensions
16      {
17          /// <summary>
18          /// <para>
19          /// Creates the sequences.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <typeparam name="TLink">
24          /// <para>The link.</para>
25          /// <para></para>
26          /// </typeparam>
27          /// <param name="sequences">
28          /// <para>The sequences.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="groupedSequence">
32          /// <para>The grouped sequence.</para>
33          /// <para></para>
34          /// </param>
35          /// <returns>
36          /// <para>The link</para>
37          /// <para></para>
38          /// </returns>
39          [MethodImpl(MethodImplOptions.AggressiveInlining)]
40          public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
            ↪ groupedSequence)
41          {
42              var finalSequence = new TLink[groupedSequence.Count];
43              for (var i = 0; i < finalSequence.Length; i++)
44              {
45                  var part = groupedSequence[i];
46                  finalSequence[i] = part.Length == 1 ? part[0] :
                    ↪ sequences.Create(part.ShiftRight());
47              }
48              return sequences.Create(finalSequence.ShiftRight());
49          }
50
51          /// <summary>
52          /// <para>
53          /// Returns the list using the specified sequences.
54          /// </para>
55          /// <para></para>
56          /// </summary>
57          /// <typeparam name="TLink">
```

```
58          /// <para>The link.</para>
59          /// <para></para>
60          /// </typeparam>
61          /// <param name="sequences">
62          /// <para>The sequences.</para>
63          /// <para></para>
64          /// </param>
65          /// <param name="sequence">
66          /// <para>The sequence.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The list.</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
75          {
76              var list = new List<TLink>();
77              var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
78              sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                ↪  LinkAddress<TLink>(sequence));
79              return list;
80          }
81      }
82  }
```

## 1.45 ./csharp/Platform.Data.Doublets.Sequences/SequencesOptions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4   using Platform.Collections.Stacks;
5   using Platform.Converters;
6   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8   using Platform.Data.Doublets.Sequences.Converters;
9   using Platform.Data.Doublets.Sequences.Walkers;
10  using Platform.Data.Doublets.Sequences.Indexes;
11  using Platform.Data.Doublets.Sequences.CriterionMatchers;
12  using System.Runtime.CompilerServices;
13
14  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16  namespace Platform.Data.Doublets.Sequences
17  {
18      /// <summary>
19      /// <para>
20      /// Represents the sequences options.
21      /// </para>
22      /// <para></para>
23      /// </summary>
24      public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪  ILinks<TLink> must contain GetConstants function.
25      {
26          /// <summary>
27          /// <para>
28          /// The default.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
33
34          /// <summary>
35          /// <para>
36          /// Gets or sets the sequence marker link value.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          public TLink SequenceMarkerLink
41          {
42              [MethodImpl(MethodImplOptions.AggressiveInlining)]
43              get;
44              [MethodImpl(MethodImplOptions.AggressiveInlining)]
45              set;
46          }
47
48          /// <summary>
49          /// <para>
```

```
        /// Gets or sets the use cascade update value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseCascadeUpdate
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use cascade delete value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseCascadeDelete
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use index value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseIndex
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        } // TODO: Update Index on sequence update/delete.

        /// <summary>
        /// <para>
        /// Gets or sets the use sequence marker value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseSequenceMarker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use compression value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseCompression
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use garbage collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseGarbageCollection
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
```

```
        }

        /// <summary>
        /// <para>
        /// Gets or sets the enforce single sequence version on write based on existing value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the enforce single sequence version on write based on new value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the marked sequence matcher value.
        /// </para>
        /// <para></para>
        /// </summary>
        public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the links to sequence converter value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the index value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISequenceIndex<TLink> Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the walker value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISequenceWalker<TLink> Walker
```

```
209         {
210             [MethodImpl(MethodImplOptions.AggressiveInlining)]
211             get;
212             [MethodImpl(MethodImplOptions.AggressiveInlining)]
213             set;
214         }
215 
216         /// <summary>
217         /// <para>
218         /// Gets or sets the read full sequence value.
219         /// </para>
220         /// <para></para>
221         /// </summary>
222         public bool ReadFullSequence
223         {
224             [MethodImpl(MethodImplOptions.AggressiveInlining)]
225             get;
226             [MethodImpl(MethodImplOptions.AggressiveInlining)]
227             set;
228         }
229 
230         // TODO: Реализовать компактификацию при чтении
231         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
232         //public bool UseRequestMarker { get; set; }
233         //public bool StoreRequestResults { get; set; }
234 
235         /// <summary>
236         /// <para>
237         /// Inits the options using the specified links.
238         /// </para>
239         /// <para></para>
240         /// </summary>
241         /// <param name="links">
242         /// <para>The links.</para>
243         /// <para></para>
244         /// </param>
245         /// <exception cref="InvalidOperationException">
246         /// <para>Cannot recreate sequence marker link.</para>
247         /// <para></para>
248         /// </exception>
249         [MethodImpl(MethodImplOptions.AggressiveInlining)]
250         public void InitOptions(ISynchronizedLinks<TLink> links)
251         {
252             if (UseSequenceMarker)
253             {
254                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
255                 {
256                     SequenceMarkerLink = links.CreatePoint();
257                 }
258                 else
259                 {
260                     if (!links.Exists(SequenceMarkerLink))
261                     {
262                         var link = links.CreatePoint();
263                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
264                         {
265                             throw new InvalidOperationException("Cannot recreate sequence marker
                                ↪  link.");
266                         }
267                     }
268                 }
269                 if (MarkedSequenceMatcher == null)
270                 {
271                     MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                        ↪  SequenceMarkerLink);
272                 }
273             }
274             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
275             if (UseCompression)
276             {
277                 if (LinksToSequenceConverter == null)
278                 {
279                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
280                     if (UseSequenceMarker)
281                     {
282                         totalSequenceSymbolFrequencyCounter = new
                            ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪  MarkedSequenceMatcher);
```

```
                    }
                    else
                    {
                        totalSequenceSymbolFrequencyCounter = new
                        ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links);
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLink>(links,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLink>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
            }
        }

        /// <summary>
        /// <para>
        /// Validates the options.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="NotSupportedException">
        /// <para>To use garbage collection UseSequenceMarker option must be on.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateOptions()
        {
            if (UseGarbageCollection && !UseSequenceMarker)
            {
                throw new NotSupportedException("To use garbage collection UseSequenceMarker
                ↪  option must be on.");
            }
        }
    }
}
```

## 1.46 ./csharp/Platform.Data.Doublets.Sequences/Time/DateTimeToLongRawNumberSequenceConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Time
{
    /// <summary>
    /// <para>
    /// Represents the date time to long raw number sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{DateTime, TLink}"/>
    public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
    {
        /// <summary>
        /// <para>
        /// The int 64 to long raw number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;

        /// <summary>
```

```
        /// <para>
        /// Initializes a new <see cref="DateTimeToLongRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="int64ToLongRawNumberConverter">
        /// <para>A int 64 to long raw number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
        ↪  int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
        ↪  int64ToLongRawNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(DateTime source) =>
        ↪  _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
    }
}
```

## 1.47 ./csharp/Platform.Data.Doublets.Sequences/Time/LongRawNumberSequenceToDateTimeConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Time
{
    /// <summary>
    /// <para>
    /// Represents the long raw number sequence to date time converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TLink, DateTime}"/>
    public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
    {
        /// <summary>
        /// <para>
        /// The long raw number converter to int 64.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LongRawNumberSequenceToDateTimeConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="longRawNumberConverterToInt64">
        /// <para>A long raw number converter to int 64.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
        ↪  longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
        ↪  longRawNumberConverterToInt64;

        /// <summary>
        /// <para>
        /// Converts the source.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The date time</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DateTime Convert(TLink source) =>
        ↪  DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
    }
}
```

## 1.48 ./csharp/Platform.Data.Doublets.Sequences/UInt64LinksExtensions.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Singletons;
using Platform.Data.Doublets.Unicode;

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

namespace Platform.Data.Doublets
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class UInt64LinksExtensions
    {
        /// <summary>
        /// <para>
        /// Uses the unicode using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
    }
}
```

## 1.49 ./csharp/Platform.Data.Doublets.Sequences/Unicode/CharToUnicodeSymbolConverter.cs

```
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the char to unicode symbol converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{char, TLink}"/>
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<char, TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
        ↪  UncheckedConverter<char, TLink>.Default;
```

```

        /// <summary>
        /// <para>
        /// The address to number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink> _addressToNumberConverter;
        /// <summary>
        /// <para>
        /// The unicode symbol marker.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly TLink _unicodeSymbolMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CharToUnicodeSymbolConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolMarker">
        /// <para>A unicode symbol marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪  addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(char source)
        {
            var unaryNumber =
            ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
            return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

## 1.50 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSequenceConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the string to unicode sequence converter.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{string, TLink}"/>
    public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<string, TLink>
    {
        /// <summary>
        /// <para>
        /// The string to unicode symbol list converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
        /// <summary>
        /// <para>
        /// The unicode symbol list to sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stringToUnicodeSymbolListConverter">
        /// <para>A string to unicode symbol list converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolListToSequenceConverter">
        /// <para>A unicode symbol list to sequence converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
        ↪  unicodeSymbolListToSequenceConverter) : base(links)
        {
            _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
            _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stringToUnicodeSymbolListConverter">
        /// <para>A string to unicode symbol list converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceLinkConverter">
        /// <para>A list to sequence link converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceMarker">
        /// <para>A unicode sequence marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
87          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
            ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            ↪  unicodeSequenceMarker)
88              : this(links, stringToUnicodeSymbolListConverter, new
                ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }
89
90          /// <summary>
91          /// <para>
92          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
93          /// </para>
94          /// <para></para>
95          /// </summary>
96          /// <param name="links">
97          /// <para>A links.</para>
98          /// <para></para>
99          /// </param>
100         /// <param name="charToUnicodeSymbolConverter">
101         /// <para>A char to unicode symbol converter.</para>
102         /// <para></para>
103         /// </param>
104         /// <param name="index">
105         /// <para>A index.</para>
106         /// <para></para>
107         /// </param>
108         /// <param name="listToSequenceLinkConverter">
109         /// <para>A list to sequence link converter.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="unicodeSequenceMarker">
113         /// <para>A unicode sequence marker.</para>
114         /// <para></para>
115         /// </param>
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪  charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
            ↪  TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
118             : this(links, new
                ↪  StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
119
120         /// <summary>
121         /// <para>
122         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
123         /// </para>
124         /// <para></para>
125         /// </summary>
126         /// <param name="links">
127         /// <para>A links.</para>
128         /// <para></para>
129         /// </param>
130         /// <param name="charToUnicodeSymbolConverter">
131         /// <para>A char to unicode symbol converter.</para>
132         /// <para></para>
133         /// </param>
134         /// <param name="listToSequenceLinkConverter">
135         /// <para>A list to sequence link converter.</para>
136         /// <para></para>
137         /// </param>
138         /// <param name="unicodeSequenceMarker">
139         /// <para>A unicode sequence marker.</para>
140         /// <para></para>
141         /// </param>
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪  charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
144             : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
145
146         /// <summary>
147         /// <para>
148         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
149         /// </para>
150         /// <para></para>
151         /// </summary>
```

```
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stringToUnicodeSymbolListConverter">
        /// <para>A string to unicode symbol list converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceLinkConverter">
        /// <para>A list to sequence link converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceMarker">
        /// <para>A unicode sequence marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
        ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
            : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
            ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(string source)
        {
            var elements = _stringToUnicodeSymbolListConverter.Convert(source);
            return _unicodeSymbolListToSequenceConverter.Convert(elements);
        }
    }
}
```

## 1.51 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSymbolsListConverter.cs

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

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the string to unicode symbols list converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{string, IList{TLink}}"/>
    public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
    {
        /// <summary>
        /// <para>
        /// The char to unicode symbol converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="StringToUnicodeSymbolsListConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="charToUnicodeSymbolConverter">
        /// <para>A char to unicode symbol converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
        ↪  charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
        ↪  charToUnicodeSymbolConverter;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The elements.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> Convert(string source)
        {
            var elements = new TLink[source.Length];
            for (var i = 0; i < elements.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            return elements;
        }
    }
}
```

## 1.52 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeMap.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode map.
    /// </para>
    /// <para></para>
    /// </summary>
    public class UnicodeMap
    {
        /// <summary>
        /// <para>
        /// The first char link.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong FirstCharLink = 1;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong MapSize = 1 + char.MaxValue;

        /// <summary>
```

```
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ILinks<ulong> _links;
        /// <summary>
        /// <para>
        /// The initialized.
        /// </para>
        /// <para></para>
        /// </summary>
        private bool _initialized;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeMap"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeMap(ILinks<ulong> links) => _links = links;

        /// <summary>
        /// <para>
        /// Inits the new using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The map.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        /// <summary>
        /// <para>
        /// Inits this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Unable to initialize UTF 16 table.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
```

```
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                        ↪  table.");
                    }
                }
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        /// <summary>
        /// <para>
        /// Creates the char to link using the specified character.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

        /// <summary>
        /// <para>
        /// Creates the link to char using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The char</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        /// <summary>
        /// <para>
        /// Determines whether is char link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

        /// <summary>
        /// <para>
        /// Creates the links to string using the specified links list.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksList">
        /// <para>The links list.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
197             /// <para>The string</para>
198             /// <para></para>
199             /// </returns>
200             [MethodImpl(MethodImplOptions.AggressiveInlining)]
201             public static string FromLinksToString(IList<ulong> linksList)
202             {
203                 var sb = new StringBuilder();
204                 for (int i = 0; i < linksList.Count; i++)
205                 {
206                     sb.Append(FromLinkToChar(linksList[i]));
207                 }
208                 return sb.ToString();
209             }
210 
211             /// <summary>
212             /// <para>
213             /// Creates the sequence link to string using the specified link.
214             /// </para>
215             /// <para></para>
216             /// </summary>
217             /// <param name="link">
218             /// <para>The link.</para>
219             /// <para></para>
220             /// </param>
221             /// <param name="links">
222             /// <para>The links.</para>
223             /// <para></para>
224             /// </param>
225             /// <returns>
226             /// <para>The string</para>
227             /// <para></para>
228             /// </returns>
229             [MethodImpl(MethodImplOptions.AggressiveInlining)]
230             public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
231             {
232                 var sb = new StringBuilder();
233                 if (links.Exists(link))
234                 {
235                     StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
236                         x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                            ↪  element =>
237                         {
238                             sb.Append(FromLinkToChar(element));
239                             return true;
240                         });
241                 }
242                 return sb.ToString();
243             }
244 
245             /// <summary>
246             /// <para>
247             /// Creates the chars to link array using the specified chars.
248             /// </para>
249             /// <para></para>
250             /// </summary>
251             /// <param name="chars">
252             /// <para>The chars.</para>
253             /// <para></para>
254             /// </param>
255             /// <returns>
256             /// <para>The ulong array</para>
257             /// <para></para>
258             /// </returns>
259             [MethodImpl(MethodImplOptions.AggressiveInlining)]
260             public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
                ↪  chars.Length);
261 
262             /// <summary>
263             /// <para>
264             /// Creates the chars to link array using the specified chars.
265             /// </para>
266             /// <para></para>
267             /// </summary>
268             /// <param name="chars">
269             /// <para>The chars.</para>
270             /// <para></para>
271             /// </param>
272             /// <param name="count">
```

```
        /// <para>The count.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The links sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromCharsToLinkArray(char[] chars, int count)
        {
            // char array to ulong array
            var linksSequence = new ulong[count];
            for (var i = 0; i < count; i++)
            {
                linksSequence[i] = FromCharToLink(chars[i]);
            }
            return linksSequence;
        }

        /// <summary>
        /// <para>
        /// Creates the string to link array using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The links sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromStringToLinkArray(string sequence)
        {
            // char array to ulong array
            var linksSequence = new ulong[sequence.Length];
            for (var i = 0; i < sequence.Length; i++)
            {
                linksSequence[i] = FromCharToLink(sequence[i]);
            }
            return linksSequence;
        }

        /// <summary>
        /// <para>
        /// Creates the string to link array groups using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                       currentCategory ==
                       ↪  CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
```

```
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Creates the link array to link array groups using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
                    var currentCategory =
                    ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                           array[absoluteLength] <= LastCharLink &&
                           currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                           ↪  array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

## 1.53 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using System.Text;

```

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode sequence to string converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink, string}"/>
    public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink, string>
    {
        /// <summary>
        /// <para>
        /// The unicode sequence criterion matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        /// <summary>
        /// <para>
        /// The sequence walker.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        /// <summary>
        /// <para>
        /// The unicode symbol to char converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSequenceToStringConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceCriterionMatcher">
        /// <para>A unicode sequence criterion matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceWalker">
        /// <para>A sequence walker.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolToCharConverter">
        /// <para>A unicode symbol to char converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
        ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
        ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
        }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
```

```
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a unicode sequence.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Convert(TLink source)
        {
            if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode sequence.");
            }
            var sequence = _links.GetSource(source);
            var sb = new StringBuilder();
            foreach(var character in _sequenceWalker.Walk(sequence))
            {
                sb.Append(_unicodeSymbolToCharConverter.Convert(character));
            }
            return sb.ToString();
        }
    }
}
```

## 1.54 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode symbol to char converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink, char}"/>
    public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink, char>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
        ↪  UncheckedConverter<TLink, char>.Default;

        /// <summary>
        /// <para>
        /// The number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IConverter<TLink> _numberToAddressConverter;
        /// <summary>
        /// <para>
        /// The unicode symbol criterion matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolToCharConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="numberToAddressConverter">
        /// <para>A number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolCriterionMatcher">
        /// <para>A unicode symbol criterion matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
        ↪  numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
        ↪  base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a unicode symbol.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The char</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public char Convert(TLink source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode symbol.");
            }
            return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
            ↪  ource(source)));
        }
    }
}
```

## 1.55 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode symbols list to unicode sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{IList{TLink}, TLink}"/>
    public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>, TLink>
    {
        /// <summary>
        /// <para>
        /// The index.
        /// </para>
```

```
24          /// <para></para>
25          /// </summary>
26          private readonly ISequenceIndex<TLink> _index;
27          /// <summary>
28          /// <para>
29          /// The list to sequence link converter.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
34          /// <summary>
35          /// <para>
36          /// The unicode sequence marker.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          private readonly TLink _unicodeSequenceMarker;
41
42          /// <summary>
43          /// <para>
44          /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
            ↪  instance.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="links">
49          /// <para>A links.</para>
50          /// <para></para>
51          /// </param>
52          /// <param name="index">
53          /// <para>A index.</para>
54          /// <para></para>
55          /// </param>
56          /// <param name="listToSequenceLinkConverter">
57          /// <para>A list to sequence link converter.</para>
58          /// <para></para>
59          /// </param>
60          /// <param name="unicodeSequenceMarker">
61          /// <para>A unicode sequence marker.</para>
62          /// <para></para>
63          /// </param>
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            ↪  ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
66          {
67              _index = index;
68              _listToSequenceLinkConverter = listToSequenceLinkConverter;
69              _unicodeSequenceMarker = unicodeSequenceMarker;
70          }
71
72          /// <summary>
73          /// <para>
74          /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
            ↪  instance.
75          /// </para>
76          /// <para></para>
77          /// </summary>
78          /// <param name="links">
79          /// <para>A links.</para>
80          /// <para></para>
81          /// </param>
82          /// <param name="listToSequenceLinkConverter">
83          /// <para>A list to sequence link converter.</para>
84          /// <para></para>
85          /// </param>
86          /// <param name="unicodeSequenceMarker">
87          /// <para>A unicode sequence marker.</para>
88          /// <para></para>
89          /// </param>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            ↪  unicodeSequenceMarker)
92              : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
                ↪  unicodeSequenceMarker) { }
93
94          /// <summary>
```

```
        /// <para>
        /// Converts the list.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="list">
        /// <para>The list.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(IList<TLink> list)
        {
            _index.Add(list);
            var sequence = _listToSequenceLinkConverter.Convert(list);
            return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

## 1.56 ./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Defines the sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ISequenceWalker<TLink>
    {
        /// <summary>
        /// <para>
        /// Walks the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerable<TLink> Walk(TLink sequence);
    }
}
```

## 1.57 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the left sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLink}"/>
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        /// <summary>
```

```
20          /// <para>
21          /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          /// <param name="links">
26          /// <para>A links.</para>
27          /// <para></para>
28          /// </param>
29          /// <param name="stack">
30          /// <para>A stack.</para>
31          /// <para></para>
32          /// </param>
33          /// <param name="isElement">
34          /// <para>A is element.</para>
35          /// <para></para>
36          /// </param>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪  isElement) : base(links, stack, isElement) { }
39
40          /// <summary>
41          /// <para>
42          /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <param name="links">
47          /// <para>A links.</para>
48          /// <para></para>
49          /// </param>
50          /// <param name="stack">
51          /// <para>A stack.</para>
52          /// <para></para>
53          /// </param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
            ↪  links.IsPartialPoint) { }
56
57          /// <summary>
58          /// <para>
59          /// Gets the next element after pop using the specified element.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="element">
64          /// <para>The element.</para>
65          /// <para></para>
66          /// </param>
67          /// <returns>
68          /// <para>The link</para>
69          /// <para></para>
70          /// </returns>
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  _links.GetSource(element);
73
74          /// <summary>
75          /// <para>
76          /// Gets the next element after push using the specified element.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <param name="element">
81          /// <para>The element.</para>
82          /// <para></para>
83          /// </param>
84          /// <returns>
85          /// <para>The link</para>
86          /// <para></para>
87          /// </returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  _links.GetTarget(element);
90
91          /// <summary>
92          /// <para>
93          /// Walks the contents using the specified element.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var links = _links;
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

## 1.58 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs

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

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

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the leveled sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ISequenceWalker{TLink}"/>
    public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// The is element.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly Func<TLink, bool> _isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
```

```
49          /// </param>
50          /// <param name="isElement">
51          /// <para>A is element.</para>
52          /// <para></para>
53          /// </param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
            ↪  base(links) => _isElement = isElement;
56
57          /// <summary>
58          /// <para>
59          /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="links">
64          /// <para>A links.</para>
65          /// <para></para>
66          /// </param>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
            ↪  _links.IsPartialPoint;
69
70          /// <summary>
71          /// <para>
72          /// Walks the sequence.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="sequence">
77          /// <para>The sequence.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>An enumerable of t link</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
86
87          /// <summary>
88          /// <para>
89          /// Returns the array using the specified sequence.
90          /// </para>
91          /// <para></para>
92          /// </summary>
93          /// <param name="sequence">
94          /// <para>The sequence.</para>
95          /// <para></para>
96          /// </param>
97          /// <returns>
98          /// <para>The link array</para>
99          /// <para></para>
100         /// </returns>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         public TLink[] ToArray(TLink sequence)
103         {
104             var length = 1;
105             var array = new TLink[length];
106             array[0] = sequence;
107             if (_isElement(sequence))
108             {
109                 return array;
110             }
111             bool hasElements;
112             do
113             {
114                 length *= 2;
115 #if USEARRAYPOOL
116                 var nextArray = ArrayPool.Allocate<ulong>(length);
117 #else
118                 var nextArray = new TLink[length];
119 #endif
120                 hasElements = false;
121                 for (var i = 0; i < array.Length; i++)
122                 {
123                     var candidate = array[i];
124                     if (_equalityComparer.Equals(array[i], default))
```

```
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var links = _links;
                        var link = links.GetLink(candidate);
                        var linkSource = links.GetSource(link);
                        var linkTarget = links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }

        /// <summary>
        /// <para>
        /// Copies the filled elements using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <param name="filledElementsCount">
        /// <para>The filled elements count.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The final array.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
        {
            var finalArray = new TLink[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
            ArrayPool.Free(array);
#endif
            return finalArray;
        }

        /// <summary>
```

```
        /// <para>
        /// Counts the filled elements using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The count.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

## 1.59 ./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.cs

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the right sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLink}"/>
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RightSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RightSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
        ↪  stack, links.IsPartialPoint) { }

        /// <summary>
        /// <para>
        /// Gets the next element after pop using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPop(TLink element) =>
        ↪  _links.GetTarget(element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLink GetNextElementAfterPush(TLink element) =>
        ↪  _links.GetSource(element);

        /// <summary>
        /// <para>
        /// Walks the contents using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLink> WalkContents(TLink element)
        {
            var parts = _links.GetLink(element);
            for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

## 1.60 ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the sequence walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ISequenceWalker{TLink}"/>
    public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
    ↪  ISequenceWalker<TLink>
    {
        /// <summary>
        /// <para>
        /// The stack.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IStack<TLink> _stack;
        /// <summary>
        /// <para>
        /// The is element.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly Func<TLink, bool> _isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint) { }

        /// <summary>
```

```
        /// <para>
        /// Walks the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elementLink">
        /// <para>The element link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);

        /// <summary>
        /// <para>
        /// Gets the next element after pop using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
```

```
156             /// </returns>
157             [MethodImpl(MethodImplOptions.AggressiveInlining)]
158             protected abstract TLink GetNextElementAfterPop(TLink element);
159 
160             /// <summary>
161             /// <para>
162             /// Gets the next element after push using the specified element.
163             /// </para>
164             /// <para></para>
165             /// </summary>
166             /// <param name="element">
167             /// <para>The element.</para>
168             /// <para></para>
169             /// </param>
170             /// <returns>
171             /// <para>The link</para>
172             /// <para></para>
173             /// </returns>
174             [MethodImpl(MethodImplOptions.AggressiveInlining)]
175             protected abstract TLink GetNextElementAfterPush(TLink element);
176 
177             /// <summary>
178             /// <para>
179             /// Walks the contents using the specified element.
180             /// </para>
181             /// <para></para>
182             /// </summary>
183             /// <param name="element">
184             /// <para>The element.</para>
185             /// <para></para>
186             /// </param>
187             /// <returns>
188             /// <para>An enumerable of t link</para>
189             /// <para></para>
190             /// </returns>
191             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192             protected abstract IEnumerable<TLink> WalkContents(TLink element);
193         }
194     }
```

## 1.61 ./csharp/Platform.Data.Doublets.Sequences.Tests/BigIntegerConvertersTests.cs

```
1   using System.Collections.Generic;
2   using System.Numerics;
3   using Platform.Data.Doublets.Memory;
4   using Platform.Data.Doublets.Memory.United.Generic;
5   using Platform.Data.Doublets.Numbers.Raw;
6   using Platform.Data.Doublets.Sequences.Converters;
7   using Platform.Data.Numbers.Raw;
8   using Platform.Memory;
9   using Xunit;
10  using TLink = System.UInt64;
11 
12  namespace Platform.Data.Doublets.Sequences.Tests
13  {
14      /// <summary>
15      /// <para>
16      /// Represents the big integer converters tests.
17      /// </para>
18      /// <para></para>
19      /// </summary>
20      public class BigIntegerConvertersTests
21      {
22          /// <summary>
23          /// <para>
24          /// Creates the links.
25          /// </para>
26          /// <para></para>
27          /// </summary>
28          /// <returns>
29          /// <para>A links of t link</para>
30          /// <para></para>
31          /// </returns>
32          public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
33 
34          /// <summary>
35          /// <para>
36          /// Creates the links using the specified data db filename.
37          /// </para>
38          /// <para></para>
```

```
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dataDbFilename">
        /// <para>The data db filename.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
            ↪  true);
            return new UnitedMemoryLinks<TLink>(new
            ↪  FileMappedResizableDirectMemory(dataDbFilename),
            ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }

        /// <summary>
        /// <para>
        /// Tests that decimal max value test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void DecimalMaxValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MaxValue);
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        /// <summary>
        /// <para>
        /// Tests that decimal min value test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void DecimalMinValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MinValue);
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        /// <summary>
        /// <para>
```

```
104         /// Tests that zero value test.
105         /// </para>
106         /// <para></para>
107         /// </summary>
108         [Fact]
109         public void ZeroValueTest()
110         {
111             var links = CreateLinks();
112             BigInteger bigInteger = new(0);
113             TLink negativeNumberMarker = links.Create();
114             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
115             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
116             BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
117             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
                ↪  negativeNumberMarker);
118             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
119             var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
120             var bigIntFromSequence =
                ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
121             Assert.Equal(bigInteger, bigIntFromSequence);
122         }
123
124         /// <summary>
125         /// <para>
126         /// Tests that one value test.
127         /// </para>
128         /// <para></para>
129         /// </summary>
130         [Fact]
131         public void OneValueTest()
132         {
133             var links = CreateLinks();
134             BigInteger bigInteger = new(1);
135             TLink negativeNumberMarker = links.Create();
136             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
137             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
138             BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
139             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
                ↪  negativeNumberMarker);
140             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
141             var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
142             var bigIntFromSequence =
                ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
143             Assert.Equal(bigInteger, bigIntFromSequence);
144         }
145     }
146 }
```

## 1.62 ./csharp/Platform.Data.Doublets.Sequences.Tests/DefaultSequenceAppenderTests.cs

```
1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3  using Platform.Data.Doublets.Memory;
4  using Platform.Data.Doublets.Memory.United.Generic;
5  using Platform.Data.Doublets.Sequences;
6  using Platform.Data.Doublets.Sequences.HeightProviders;
7  using Platform.Data.Numbers.Raw;
8  using Platform.Interfaces;
9  using Platform.Memory;
10 using Platform.Numbers;
11 using Xunit;
12 using Xunit.Abstractions;
13 using TLink = System.UInt64;
14
15 namespace Platform.Data.Doublets.Sequences.Tests
16 {
17     /// <summary>
18     /// <para>
19     /// Represents the default sequence appender tests.
20     /// </para>
21     /// <para></para>
22     /// </summary>
23     public class DefaultSequenceAppenderTests
24     {
25         /// <summary>
26         /// <para>
```

```
        /// The output.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly ITestOutputHelper _output;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceAppenderTests"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="output">
        /// <para>A output.</para>
        /// <para></para>
        /// </param>
        public DefaultSequenceAppenderTests(ITestOutputHelper output)
        {
            _output = output;
        }
        /// <summary>
        /// <para>
        /// Creates the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

        /// <summary>
        /// <para>
        /// Creates the links using the specified data db filename.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dataDBFilename">
        /// <para>The data db filename.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
            ↪  true);
            return new UnitedMemoryLinks<TLink>(new
            ↪  FileMappedResizableDirectMemory(dataDBFilename),
            ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }

        /// <summary>
        /// <para>
        /// Represents the value criterion matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="ICriterionMatcher{TLink}"/>
        public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
        {
            /// <summary>
            /// <para>
            /// The links.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly ILinks<TLink> Links;
            /// <summary>
            /// <para>
```

```
            /// The marker.
            /// </para>
            /// <para></para>
            /// </summary>
            public readonly TLink Marker;
            /// <summary>
            /// <para>
            /// Initializes a new <see cref="ValueCriterionMatcher"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="marker">
            /// <para>A marker.</para>
            /// <para></para>
            /// </param>
            public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
            {
                Links = links;
                Marker = marker;
            }

            /// <summary>
            /// <para>
            /// Determines whether this instance is matched.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            public bool IsMatched(TLink link) =>
             ↪  EqualityComparer<TLink>.Default.Equals(Links.GetSource(link), Marker);
        }

        /// <summary>
        /// <para>
        /// Tests that append array bug.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void AppendArrayBug()
        {
            ILinks<TLink> links = CreateLinks();
            TLink zero = default;
            var markerIndex = Arithmetic.Increment(zero);
            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
            var sequence = links.Create();
            sequence = links.Update(sequence, meaningRoot, sequence);
            var appendant = links.Create();
            appendant = links.Update(appendant, meaningRoot, appendant);
            ValueCriterionMatcher<TLink> valueCriterionMatcher = new(links, meaningRoot);
            DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
             ↪  new(links, valueCriterionMatcher);
            DefaultSequenceAppender<TLink> defaultSequenceAppender = new(links, new
             ↪  DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
            var newArray = defaultSequenceAppender.Append(sequence, appendant);
            var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
            Assert.Equal("(4:(2:1 2) (3:1 3))", output);
        }
    }
}
```

## 1.63 ./csharp/Platform.Data.Doublets.Sequences.Tests/ILinksExtensionsTests.cs

```
using Xunit;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
```

```
    /// <para>
    /// Represents the links extensions tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class ILinksExtensionsTests
    {
        /// <summary>
        /// <para>
        /// Tests that format test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void FormatTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var link = links.Create();
                var linkString = links.Format(link);
                Assert.Equal("(1: 1 1)", linkString);
            }
        }
    }
}
```

## 1.64 ./csharp/Platform.Data.Doublets.Sequences.Tests/OptimalVariantSequenceTests.cs

```
using System;
using System.Linq;
using Xunit;
using Platform.Collections.Stacks;
using Platform.Collections.Arrays;
using Platform.Memory;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;
using Platform.Data.Doublets.Memory;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the optimal variant sequence tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class OptimalVariantSequenceTests
    {
        /// <summary>
        /// <para>
        /// The sequence example.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        /// <summary>
        /// <para>
        /// The lorem ipsum example.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
        ↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore
        ↪  magna aliqua.
Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
Et malesuada fames ac turpis egestas sed.
Eget velit aliquet sagittis id consectetur purus.
Dignissim cras tincidunt lobortis feugiat vivamus.
Vitae aliquet nec ullamcorper sit.
Lectus quam id leo in vitae.
```

```
Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
Integer eget aliquet nibh praesent tristique.
Vitae congue eu consequat ac felis donec et odio.
Tristique et egestas quis ipsum suspendisse.
Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
Imperdiet proin fermentum leo vel orci.
In ante metus dictum at tempor commodo.
Nisi lacus sed viverra tellus in.
Quam vulputate dignissim suspendisse in.
Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
Gravida cum sociis natoque penatibus et magnis dis parturient.
Risus quis varius quam quisque id diam.
Congue nisi vitae suscipit tellus mauris a diam maecenas.
Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
Pharetra vel turpis nunc eget lorem dolor sed viverra.
Mattis pellentesque id nibh tortor id aliquet.
Purus non enim praesent elementum facilisis leo vel.
Etiam sit amet nisl purus in mollis nunc sed.
Tortor at auctor urna nunc id cursus metus aliquam.
Volutpat odio facilisis mauris sit amet.
Turpis egestas pretium aenean pharetra magna ac placerat.
Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
Porttitor leo a diam sollicitudin tempor id eu.
Volutpat sed cras ornare arcu dui.
Ut aliquam purus sit amet luctus venenatis lectus magna.
Aliquet risus feugiat in ante metus dictum at.
Mattis nunc sed blandit libero.
Elit pellentesque habitant morbi tristique senectus et netus.
Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
Diam donec adipiscing tristique risus nec feugiat.
Pulvinar mattis nunc sed blandit libero volutpat.
Cras fermentum odio eu feugiat pretium nibh ipsum.
In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
A iaculis at erat pellentesque.
Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
Eget lorem dolor sed viverra ipsum nunc.
Leo a diam sollicitudin tempor id eu.
Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";

        /// <summary>
        /// <para>
        /// Tests that links based frequency stored optimal variant sequence test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var constants = links.Constants;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var meaningRoot = links.CreatePoint();
                var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
                var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                ↪  constants.Itself);

                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                ↪  frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                ↪  frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                ↪  unaryNumberToAddressConverter);
```

```
125                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
126                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
127
128                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
                    ↪  new LeveledSequenceWalker<ulong>(links) });
129
130                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
131             }
132         }
133
134         /// <summary>
135         /// <para>
136         /// Tests that dictionary based frequency stored optimal variant sequence test.
137         /// </para>
138         /// <para></para>
139         /// </summary>
140         [Fact]
141         public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
142         {
143             using (var scope = new TempLinksTestScope(useSequences: false))
144             {
145                 var links = scope.Links;
146
147                 links.UseUnicode();
148
149                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
150
151                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
152
153                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
154
155                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
156                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
157
158                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
159                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
160
161                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
                    ↪  new LeveledSequenceWalker<ulong>(links) });
162
163                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
164             }
165         }
166
167         /// <summary>
168         /// <para>
169         /// Executes the test using the specified sequences.
170         /// </para>
171         /// <para></para>
172         /// </summary>
173         /// <param name="sequences">
174         /// <para>The sequences.</para>
175         /// <para></para>
176         /// </param>
177         /// <param name="sequence">
178         /// <para>The sequence.</para>
179         /// <para></para>
180         /// </param>
181         /// <param name="sequenceToItsLocalElementLevelsConverter">
182         /// <para>The sequence to its local element levels converter.</para>
183         /// <para></para>
184         /// </param>
185         /// <param name="index">
186         /// <para>The index.</para>
187         /// <para></para>
188         /// </param>
```

```
189             /// <param name="optimalVariantConverter">
190             /// <para>The optimal variant converter.</para>
191             /// <para></para>
192             /// </param>
193             private static void ExecuteTest(Sequences sequences, ulong[] sequence,
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>
                ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
                ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
194             {
195                 index.Add(sequence);
196
197                 var optimalVariant = optimalVariantConverter.Convert(sequence);
198
199                 var readSequence1 = sequences.ToList(optimalVariant);
200
201                 Assert.True(sequence.SequenceEqual(readSequence1));
202             }
203
204             /// <summary>
205             /// <para>
206             /// Tests that saved sequences optimization test.
207             /// </para>
208             /// <para></para>
209             /// </summary>
210             [Fact]
211             public static void SavedSequencesOptimizationTest()
212             {
213                 LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                    ↪  (long.MaxValue + 1UL, ulong.MaxValue));
214
215                 using (var memory = new HeapResizableDirectMemory())
216                 using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
                    ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
217                 {
218                     var links = new UInt64Links(disposableLinks);
219
220                     var root = links.CreatePoint();
221
222                     //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
223                     var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
224
225                     var unicodeSymbolMarker = links.GetOrCreate(root,
                        ↪  addressToNumberConverter.Convert(1));
226                     var unicodeSequenceMarker = links.GetOrCreate(root,
                        ↪  addressToNumberConverter.Convert(2));
227
228                     var totalSequenceSymbolFrequencyCounter = new
                        ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
229                     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                        ↪  totalSequenceSymbolFrequencyCounter);
230                     var index = new
                        ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
231                     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                        ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
232                     var sequenceToItsLocalElementLevelsConverter = new
                        ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                        ↪  linkToItsFrequencyNumberConverter);
233                     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                        ↪  sequenceToItsLocalElementLevelsConverter);
234
235                     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                        ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
236
237                     var unicodeSequencesOptions = new SequencesOptions<ulong>()
238                     {
239                         UseSequenceMarker = true,
240                         SequenceMarkerLink = unicodeSequenceMarker,
241                         UseIndex = true,
242                         Index = index,
243                         LinksToSequenceConverter = optimalVariantConverter,
244                         Walker = walker,
245                         UseGarbageCollection = true
246                     };
247
248                     var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
                        ↪  unicodeSequencesOptions);
249
250                     // Create some sequences
```

```
                var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
                var arrays = strings.Select(x => x.Select(y =>
                ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

                // get list of sequences links
                // for each sequence link
                //   create new sequence version
                //   if new sequence is not the same as sequence link
                //     delete sequence link
                //     collect garbadge
                unicodeSequences.CompactAll();

                var linksCountAfterCompactification = links.Count();

                Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
            }
        }
    }
}
```

## 1.65 ./csharp/Platform.Data.Doublets.Sequences.Tests/RationalNumbersTests.cs

```
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Numbers.Rational;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Xunit;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the rational numbers tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class RationalNumbersTests
    {
        /// <summary>
        /// <para>
        /// Creates the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

        /// <summary>
        /// <para>
        /// Creates the links using the specified data db filename.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dataDbFilename">
        /// <para>The data db filename.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
```

```
52          {
53              var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                ↪  true);
54              return new UnitedMemoryLinks<TLink>(new
                ↪  FileMappedResizableDirectMemory(dataDbFilename),
                ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
55          }
56
57          /// <summary>
58          /// <para>
59          /// Tests that decimal min value test.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          [Fact]
64          public void DecimalMinValueTest()
65          {
66              const decimal @decimal = decimal.MinValue;
67              var links = CreateLinks();
68              TLink negativeNumberMarker = links.Create();
69              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
70              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
71              BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
72              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
73              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
74              DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
75              RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
76              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
77              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
78             Assert.Equal(@decimal, decimalFromRational);
79          }
80
81          /// <summary>
82          /// <para>
83          /// Tests that decimal max value test.
84          /// </para>
85          /// <para></para>
86          /// </summary>
87          [Fact]
88          public void DecimalMaxValueTest()
89          {
90              const decimal @decimal = decimal.MaxValue;
91              var links = CreateLinks();
92              TLink negativeNumberMarker = links.Create();
93              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
94              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
95              BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
96              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
97              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
98              DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
99              RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
100             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
101             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
102             Assert.Equal(@decimal, decimalFromRational);
103         }
104
105         /// <summary>
106         /// <para>
107         /// Tests that decimal positive half test.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         [Fact]
112         public void DecimalPositiveHalfTest()
113         {
114             const decimal @decimal = 0.5M;
115             var links = CreateLinks();
```

```
116             TLink negativeNumberMarker = links.Create();
117             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
118             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
119             BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
120             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
121             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
122             DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
123             RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
124             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
125             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
126             Assert.Equal(@decimal, decimalFromRational);
127         }
128
129         /// <summary>
130         /// <para>
131         /// Tests that decimal negative half test.
132         /// </para>
133         /// <para></para>
134         /// </summary>
135         [Fact]
136         public void DecimalNegativeHalfTest()
137         {
138             const decimal @decimal = -0.5M;
139             var links = CreateLinks();
140             TLink negativeNumberMarker = links.Create();
141             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
142             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
143             BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
144             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
145             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
146             DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
147             RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
148             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
149             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
150             Assert.Equal(@decimal, decimalFromRational);
151         }
152
153         /// <summary>
154         /// <para>
155         /// Tests that decimal one test.
156         /// </para>
157         /// <para></para>
158         /// </summary>
159         [Fact]
160         public void DecimalOneTest()
161         {
162             const decimal @decimal = 1;
163             var links = CreateLinks();
164             TLink negativeNumberMarker = links.Create();
165             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
166             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
167             BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
168             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
169             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
170             DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
171             RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
172             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
173             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
174             Assert.Equal(@decimal, decimalFromRational);
175         }
176
177         /// <summary>
178         /// <para>
```

```
179         /// Tests that decimal minus one test.
180         /// </para>
181         /// <para></para>
182         /// </summary>
183         [Fact]
184         public void DecimalMinusOneTest()
185         {
186             const decimal @decimal = -1;
187             var links = CreateLinks();
188             TLink negativeNumberMarker = links.Create();
189             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
190             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
191             BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
192             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
193             RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
194             DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
195             RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
196             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
197             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
198             Assert.Equal(@decimal, decimalFromRational);
199         }
200     }
201 }
```

## 1.66 ./csharp/Platform.Data.Doublets.Sequences.Tests/ReadSequenceTests.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Data.Sequences;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.Walkers;
9  using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Sequences.Tests
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the read sequence tests.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public static class ReadSequenceTests
20     {
21         /// <summary>
22         /// <para>
23         /// Tests that read sequence test.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         [Fact]
28         public static void ReadSequenceTest()
29         {
30             const long sequenceLength = 2000;
31
32             using (var scope = new TempLinksTestScope(useSequences: false))
33             {
34                 var links = scope.Links;
35                 var sequences = new Sequences(links, new SequencesOptions<ulong> { Walker = new
                    ↪  LeveledSequenceWalker<ulong>(links) });
36
37                 var sequence = new ulong[sequenceLength];
38                 for (var i = 0; i < sequenceLength; i++)
39                 {
40                     sequence[i] = links.Create();
41                 }
42
43                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
44
45                 var sw1 = Stopwatch.StartNew();
46                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
47
48                 var sw2 = Stopwatch.StartNew();
```

```
                var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var readSequence2 = new List<ulong>();
                SequenceWalker.WalkRight(balancedVariant,
                                         links.GetSource,
                                         links.GetTarget,
                                         links.IsPartialPoint,
                                         readSequence2.Add);
                sw3.Stop();

                Assert.True(sequence.SequenceEqual(readSequence1));

                Assert.True(sequence.SequenceEqual(readSequence2));

                // Assert.True(sw2.Elapsed < sw3.Elapsed);

                Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
                ↪  {sw2.Elapsed}");

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

## 1.67 ./csharp/Platform.Data.Doublets.Sequences.Tests/SequencesTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Collections;
using Platform.Collections.Arrays;
using Platform.Random;
using Platform.IO;
using Platform.Singletons;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the sequences tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class SequencesTests
    {
        /// <summary>
        /// <para>
        /// The instance.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequencesTests"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        /// <summary>
        /// <para>
        /// Tests that create all variants test.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CreateAllVariantsTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var sw1 = Stopwatch.StartNew();
                var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

                Assert.True(results1.Count > results2.Length);
                Assert.True(sw1.Elapsed > sw2.Elapsed);

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }

                Assert.True(links.Count() == 0);
            }
        }

        //[Fact]
        //public void CUDTest()
        //{
        //    var tempFilename = Path.GetTempFileName();

        //    const long sequenceLength = 8;

        //    const ulong itself = LinksConstants.Itself;

        //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
        ↪  DefaultLinksSizeStep))
        //    using (var links = new Links(memoryAdapter))
        //    {
        //        var sequence = new ulong[sequenceLength];
        //        for (var i = 0; i < sequenceLength; i++)
        //            sequence[i] = links.Create(itself, itself);

        //        SequencesOptions o = new SequencesOptions();

        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
        //        o.

        //        var sequences = new Sequences(links);

        //        var sw1 = Stopwatch.StartNew();
        //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();

        //        var sw2 = Stopwatch.StartNew();
        //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();

        //        Assert.True(results1.Count > results2.Length);
        //        Assert.True(sw1.Elapsed > sw2.Elapsed);

        //        for (var i = 0; i < sequenceLength; i++)
        //            links.Delete(sequence[i]);
        //    }

        //    File.Delete(tempFilename);
        //}

        /// <summary>
        /// <para>
```

```
        /// Tests that all variants search test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void AllVariantsSearchTest()
        {
            const long sequenceLength = 8;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();

                //for (int i = 0; i < createResults.Length; i++)
                //    sequences.Create(createResults[i]);

                var sw0 = Stopwatch.StartNew();
                var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();

                var sw1 = Stopwatch.StartNew();
                var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();

                var sw2 = Stopwatch.StartNew();
                var searchResults2 = sequences.Each1(sequence); sw2.Stop();

                var sw3 = Stopwatch.StartNew();
                var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();

                var intersection0 = createResults.Intersect(searchResults0).ToList();
                Assert.True(intersection0.Count == searchResults0.Count);
                Assert.True(intersection0.Count == createResults.Length);

                var intersection1 = createResults.Intersect(searchResults1).ToList();
                Assert.True(intersection1.Count == searchResults1.Count);
                Assert.True(intersection1.Count == createResults.Length);

                var intersection2 = createResults.Intersect(searchResults2).ToList();
                Assert.True(intersection2.Count == searchResults2.Count);
                Assert.True(intersection2.Count == createResults.Length);

                var intersection3 = createResults.Intersect(searchResults3).ToList();
                Assert.True(intersection3.Count == searchResults3.Count);
                Assert.True(intersection3.Count == createResults.Length);

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Tests that balanced variant search test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void BalancedVariantSearchTest()
        {
            const long sequenceLength = 200;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
```

```
                sequence[i] = links.Create();
            }

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

            var sw1 = Stopwatch.StartNew();
            var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();

            var sw3 = Stopwatch.StartNew();
            var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();

            // На количестве в 200 элементов это будет занимать вечность
            //var sw4 = Stopwatch.StartNew();
            //var searchResults4 = sequences.Each(sequence); sw4.Stop();

            Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);

            Assert.True(searchResults3.Count == 1 && balancedVariant ==
            ↪  searchResults3.First());

            //Assert.True(sw1.Elapsed < sw2.Elapsed);

            for (var i = 0; i < sequenceLength; i++)
            {
                links.Delete(sequence[i]);
            }
        }
    }

    /// <summary>
    /// <para>
    /// Tests that all partial variants search test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void AllPartialVariantsSearchTest()
    {
        const long sequenceLength = 8;

        using (var scope = new TempLinksTestScope(useSequences: true))
        {
            var links = scope.Links;
            var sequences = scope.Sequences;

            var sequence = new ulong[sequenceLength];
            for (var i = 0; i < sequenceLength; i++)
            {
                sequence[i] = links.Create();
            }

            var createResults = sequences.CreateAllVariants2(sequence);

            //var createResultsStrings = createResults.Select(x => x + ": " +
            ↪  sequences.FormatSequence(x)).ToList();
            //Global.Trash = createResultsStrings;

            var partialSequence = new ulong[sequenceLength - 2];

            Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);

            var sw1 = Stopwatch.StartNew();
            var searchResults1 =
            ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();

            var sw2 = Stopwatch.StartNew();
            var searchResults2 =
            ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

            //var sw3 = Stopwatch.StartNew();
            //var searchResults3 =
            ↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();

            var sw4 = Stopwatch.StartNew();
            var searchResults4 =
            ↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
```

```
280
281                 //Global.Trash = searchResults3;
282
283                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪  sequences.FormatSequence(x)).ToList();
284                 //Global.Trash = searchResults1Strings;
285
286                 var intersection1 = createResults.Intersect(searchResults1).ToList();
287                 Assert.True(intersection1.Count == createResults.Length);
288
289                 var intersection2 = createResults.Intersect(searchResults2).ToList();
290                 Assert.True(intersection2.Count == createResults.Length);
291
292                 var intersection4 = createResults.Intersect(searchResults4).ToList();
293                 Assert.True(intersection4.Count == createResults.Length);
294
295                 for (var i = 0; i < sequenceLength; i++)
296                 {
297                     links.Delete(sequence[i]);
298                 }
299             }
300         }
301
302         /// <summary>
303         /// <para>
304         /// Tests that balanced partial variants search test.
305         /// </para>
306         /// <para></para>
307         /// </summary>
308         [Fact]
309         public static void BalancedPartialVariantsSearchTest()
310         {
311             const long sequenceLength = 200;
312
313             using (var scope = new TempLinksTestScope(useSequences: true))
314             {
315                 var links = scope.Links;
316                 var sequences = scope.Sequences;
317
318                 var sequence = new ulong[sequenceLength];
319                 for (var i = 0; i < sequenceLength; i++)
320                 {
321                     sequence[i] = links.Create();
322                 }
323
324                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
325
326                 var balancedVariant = balancedVariantConverter.Convert(sequence);
327
328                 var partialSequence = new ulong[sequenceLength - 2];
329
330                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
331
332                 var sw1 = Stopwatch.StartNew();
333                 var searchResults1 =
                    ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
334
335                 var sw2 = Stopwatch.StartNew();
336                 var searchResults2 =
                    ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
337
338                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
339
340                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪  searchResults2.First());
341
342                 for (var i = 0; i < sequenceLength; i++)
343                 {
344                     links.Delete(sequence[i]);
345                 }
346             }
347         }
348
349         /// <summary>
350         /// <para>
351         /// Tests that pattern match test.
352         /// </para>
353         /// <para></para>
354         /// </summary>
```

```
355         [Fact(Skip = "Correct implementation is pending")]
356         public static void PatternMatchTest()
357         {
358             var zeroOrMany = Sequences.ZeroOrMany;
359
360             using (var scope = new TempLinksTestScope(useSequences: true))
361             {
362                 var links = scope.Links;
363                 var sequences = scope.Sequences;
364
365                 var e1 = links.Create();
366                 var e2 = links.Create();
367
368                 var sequence = new[]
369                 {
370                     e1, e2, e1, e2 // mama / papa
371                 };
372
373                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
374
375                 var balancedVariant = balancedVariantConverter.Convert(sequence);
376
377                 // 1: [1]
378                 // 2: [2]
379                 // 3: [1,2]
380                 // 4: [1,2,1,2]
381
382                 var doublet = links.GetSource(balancedVariant);
383
384                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
385
386                 Assert.True(matchedSequences1.Count == 0);
387
388                 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
389
390                 Assert.True(matchedSequences2.Count == 0);
391
392                 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
393
394                 Assert.True(matchedSequences3.Count == 0);
395
396                 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
397
398                 Assert.Contains(doublet, matchedSequences4);
399                 Assert.Contains(balancedVariant, matchedSequences4);
400
401                 for (var i = 0; i < sequence.Length; i++)
402                 {
403                     links.Delete(sequence[i]);
404                 }
405             }
406         }
407
408         /// <summary>
409         /// <para>
410         /// Tests that index test.
411         /// </para>
412         /// <para></para>
413         /// </summary>
414         [Fact]
415         public static void IndexTest()
416         {
417             using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
                 ↪ true }, useSequences: true))
418             {
419                 var links = scope.Links;
420                 var sequences = scope.Sequences;
421                 var index = sequences.Options.Index;
422
423                 var e1 = links.Create();
424                 var e2 = links.Create();
425
426                 var sequence = new[]
427                 {
428                     e1, e2, e1, e2 // mama / papa
429                 };
430
431                 Assert.False(index.MightContain(sequence));
432
433                 index.Add(sequence);
```

```
434
435                 Assert.True(index.MightContain(sequence));
436             }
437         }
438
439         /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
440         private static readonly string _exampleText =
441             @"([english
                ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
442
443 Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
    ↪  (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
    ↪  где есть место для нового начала? Разве пустота это не характеристика пространства?
    ↪  Пространство это то, что можно чем-то наполнить?
444
445 [![чёрное пространство, белое
    ↪  пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↪  ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
    ↪  Platform/master/doc/Intro/1.png)
446
447 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
    ↪  форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
448
449 [![чёрное пространство, чёрная
    ↪  точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
    ↪  ""чёрное пространство, чёрная
    ↪  точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
450
451 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↪  так? Инверсия? Отражение? Сумма?
452
453 [![белая точка, чёрная
    ↪  точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая
    ↪  точка, чёрная
    ↪  точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
454
455 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
    ↪  если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
    ↪  Гранью? Разделителем? Единицей?
456
457 [![две белые точки, чёрная вертикальная
    ↪  линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две
    ↪  белые точки, чёрная вертикальная
    ↪  линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
458
459 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
    ↪  только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
    ↪  замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
    ↪  можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
    ↪  Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
    ↪  у него нет размера? Будет ли круг точкой? Точка состоящая из точек?
460
461 [![белая вертикальная линия, чёрный
    ↪  круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая
    ↪  вертикальная линия, чёрный
    ↪  круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
462
463 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
    ↪  тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
    ↪  Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
    ↪  элементарная единица смысла?
464
465 [![белый круг, чёрная горизонтальная
    ↪  линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый
    ↪  круг, чёрная горизонтальная
    ↪  линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
466
467 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить,
    ↪  связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От
    ↪  родителя к ребёнку? От общего к частному?
468
469 [![белая горизонтальная линия, чёрная горизонтальная
    ↪  стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
    ↪  ""белая горизонтальная линия, чёрная горизонтальная
    ↪  стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
470
```

```
Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?

[![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие? Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в его конечном состоянии, если конечно конец определён направлением?

[![белая обычная и направленная связи, чёрная типизированная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая обычная и направленная связи, чёрная типизированная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри? Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

[![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?

[![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

[![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";

        /// <summary>
        /// <para>
        /// The example lorem ipsum text.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        /// <summary>
        /// <para>
        /// Tests that compression test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CompressionTest()
```

```
513         {
514             using (var scope = new TempLinksTestScope(useSequences: true))
515             {
516                 var links = scope.Links;
517                 var sequences = scope.Sequences;
518 
519                 var e1 = links.Create();
520                 var e2 = links.Create();
521 
522                 var sequence = new[]
523                 {
524                     e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
525                 };
526 
527                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
528                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
529                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
530                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
                    ↪  balancedVariantConverter, doubletFrequenciesCache);
531 
532                 var compressedVariant = compressingConverter.Convert(sequence);
533 
534                 // 1: [1]        (1->1) point
535                 // 2: [2]        (2->2) point
536                 // 3: [1,2]      (1->2) doublet
537                 // 4: [1,2,1,2] (3->3) doublet
538 
539                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
540                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
541                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
542                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
543 
544                 var source = _constants.SourcePart;
545                 var target = _constants.TargetPart;
546 
547                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
548                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
549                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
550                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
551 
552                 // 4 - length of sequence
553                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
554                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
555                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
556                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
557             }
558         }
559 
560         /// <summary>
561         /// <para>
562         /// Tests that compression efficiency test.
563         /// </para>
564         /// <para></para>
565         /// </summary>
566         [Fact]
567         public static void CompressionEfficiencyTest()
568         {
569             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                ↪  StringSplitOptions.RemoveEmptyEntries);
570             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
571             var totalCharacters = arrays.Select(x => x.Length).Sum();
572 
573             using (var scope1 = new TempLinksTestScope(useSequences: true))
574             using (var scope2 = new TempLinksTestScope(useSequences: true))
575             using (var scope3 = new TempLinksTestScope(useSequences: true))
576             {
577                 scope1.Links.Unsync.UseUnicode();
578                 scope2.Links.Unsync.UseUnicode();
579                 scope3.Links.Unsync.UseUnicode();
580 
581                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
```

```
582                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
583                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
584                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
585
586                 //var compressor2 = scope2.Sequences;
587                 var compressor3 = scope3.Sequences;
588
589                 var constants = Default<LinksConstants<ulong>>.Instance;
590
591                 var sequences = compressor3;
592                 //var meaningRoot = links.CreatePoint();
593                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
594                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
595                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
596
597                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
598                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
599                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
600                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
601                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
602                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
603
604                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
605
606                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
607
608                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
609                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
610
611                 var compressed1 = new ulong[arrays.Length];
612                 var compressed2 = new ulong[arrays.Length];
613                 var compressed3 = new ulong[arrays.Length];
614
615                 var START = 0;
616                 var END = arrays.Length;
617
618                 //for (int i = START; i < END; i++)
619                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
620
621                 var initialCount1 = scope2.Links.Unsync.Count();
622
623                 var sw1 = Stopwatch.StartNew();
624
625                 for (int i = START; i < END; i++)
626                 {
627                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
628                     compressed1[i] = compressor1.Convert(arrays[i]);
629                 }
630
631                 var elapsed1 = sw1.Elapsed;
632
633                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
634
635                 var initialCount2 = scope2.Links.Unsync.Count();
636
637                 var sw2 = Stopwatch.StartNew();
638
639                 for (int i = START; i < END; i++)
640                 {
```

```
641                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
642                 }
643
644                 var elapsed2 = sw2.Elapsed;
645
646                 for (int i = START; i < END; i++)
647                 {
648                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
649                 }
650
651                 var initialCount3 = scope3.Links.Unsync.Count();
652
653                 var sw3 = Stopwatch.StartNew();
654
655                 for (int i = START; i < END; i++)
656                 {
657                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
658                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
659                 }
660
661                 var elapsed3 = sw3.Elapsed;
662
663                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
664
665                 // Assert.True(elapsed1 > elapsed2);
666
667                 // Checks
668                 for (int i = START; i < END; i++)
669                 {
670                     var sequence1 = compressed1[i];
671                     var sequence2 = compressed2[i];
672                     var sequence3 = compressed3[i];
673
674                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
675
676                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
677
678                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
679
680                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
681                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
682                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
683
684                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
685                     //    Assert.False(structure1 == structure2);
686                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
687                     //    Assert.False(structure3 == structure2);
688
689                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
690                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
691                 }
692
693                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
694                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
695                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
696
697                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
698
699                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
700                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
701
```

```
                var duplicateProvider1 = new DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
                var duplicateProvider2 = new DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
                var duplicateProvider3 = new DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);

                var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
                var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
                var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);

                var duplicates1 = duplicateCounter1.Count();

                ConsoleHelpers.Debug("------");

                var duplicates2 = duplicateCounter2.Count();

                ConsoleHelpers.Debug("------");

                var duplicates3 = duplicateCounter3.Count();

                Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");

                linkFrequenciesCache1.ValidateFrequencies();
                linkFrequenciesCache3.ValidateFrequencies();
            }
        }

        /// <summary>
        /// <para>
        /// Tests that compression stability test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CompressionStabilityTest()
        {
            // TODO: Fix bug (do a separate test)
            //const ulong minNumbers = 0;
            //const ulong maxNumbers = 1000;

            const ulong minNumbers = 10000;
            const ulong maxNumbers = 12500;

            var strings = new List<string>();

            for (ulong i = minNumbers; i < maxNumbers; i++)
            {
                strings.Add(i.ToString());
            }

            var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
            var totalCharacters = arrays.Select(x => x.Length).Sum();

            using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new SequencesOptions<ulong> { UseCompression = true, EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
            using (var scope2 = new TempLinksTestScope(useSequences: true))
            {
                scope1.Links.UseUnicode();
                scope2.Links.UseUnicode();

                //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
                var compressor1 = scope1.Sequences;
                var compressor2 = scope2.Sequences;

                var compressed1 = new ulong[arrays.Length];
                var compressed2 = new ulong[arrays.Length];

                var sw1 = Stopwatch.StartNew();

                var START = 0;
                var END = arrays.Length;

                // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
                // Stability issue starts at 10001 or 11000
                //for (int i = START; i < END; i++)
                //{
                //    var first = compressor1.Compress(arrays[i]);
```

```
776                     //    var second = compressor1.Compress(arrays[i]);
777
778                     //    if (first == second)
779                     //        compressed1[i] = first;
780                     //    else
781                     //    {
782                     //        // TODO: Find a solution for this case
783                     //    }
784                     //}
785
786                     for (int i = START; i < END; i++)
787                     {
788                         var first = compressor1.Create(arrays[i].ShiftRight());
789                         var second = compressor1.Create(arrays[i].ShiftRight());
790
791                         if (first == second)
792                         {
793                             compressed1[i] = first;
794                         }
795                         else
796                         {
797                             // TODO: Find a solution for this case
798                         }
799                     }
800
801                     var elapsed1 = sw1.Elapsed;
802
803                     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
804
805                     var sw2 = Stopwatch.StartNew();
806
807                     for (int i = START; i < END; i++)
808                     {
809                         var first = balancedVariantConverter.Convert(arrays[i]);
810                         var second = balancedVariantConverter.Convert(arrays[i]);
811
812                         if (first == second)
813                         {
814                             compressed2[i] = first;
815                         }
816                     }
817
818                     var elapsed2 = sw2.Elapsed;
819
820                     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
821
822                     Assert.True(elapsed1 > elapsed2);
823
824                     // Checks
825                     for (int i = START; i < END; i++)
826                     {
827                         var sequence1 = compressed1[i];
828                         var sequence2 = compressed2[i];
829
830                         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
831                         {
832                             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
833
834                             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
835
836                             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪  link.IsPartialPoint());
837                             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪  link.IsPartialPoint());
838
839                             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪  arrays[i].Length > 3)
840                             //    Assert.False(structure1 == structure2);
841
842                             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
843                         }
844                     }
845
846                     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
847                     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
848
```

```
849                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
850
851                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
852
853                 //compressor1.ValidateFrequencies();
854             }
855         }
856
857         /// <summary>
858         /// <para>
859         /// Tests that rundom numbers compression quality test.
860         /// </para>
861         /// <para></para>
862         /// </summary>
863         [Fact]
864         public static void RundomNumbersCompressionQualityTest()
865         {
866             const ulong N = 500;
867
868             //const ulong minNumbers = 10000;
869             //const ulong maxNumbers = 20000;
870
871             //var strings = new List<string>();
872
873             //for (ulong i = 0; i < N; i++)
874             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                    ↪  maxNumbers).ToString());
875
876             var strings = new List<string>();
877
878             for (ulong i = 0; i < N; i++)
879             {
880                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
881             }
882
883             strings = strings.Distinct().ToList();
884
885             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
886             var totalCharacters = arrays.Select(x => x.Length).Sum();
887
888             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                    ↪  SequencesOptions<ulong> { UseCompression = true,
                    ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
889             using (var scope2 = new TempLinksTestScope(useSequences: true))
890             {
891                 scope1.Links.UseUnicode();
892                 scope2.Links.UseUnicode();
893
894                 var compressor1 = scope1.Sequences;
895                 var compressor2 = scope2.Sequences;
896
897                 var compressed1 = new ulong[arrays.Length];
898                 var compressed2 = new ulong[arrays.Length];
899
900                 var sw1 = Stopwatch.StartNew();
901
902                 var START = 0;
903                 var END = arrays.Length;
904
905                 for (int i = START; i < END; i++)
906                 {
907                     compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
908                 }
909
910                 var elapsed1 = sw1.Elapsed;
911
912                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
913
914                 var sw2 = Stopwatch.StartNew();
915
916                 for (int i = START; i < END; i++)
917                 {
918                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
919                 }
920
921                 var elapsed2 = sw2.Elapsed;
922
```

```
923                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
924
925                 Assert.True(elapsed1 > elapsed2);
926
927                 // Checks
928                 for (int i = START; i < END; i++)
929                 {
930                     var sequence1 = compressed1[i];
931                     var sequence2 = compressed2[i];
932
933                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
934                     {
935                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
936
937                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
938
939                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
940                     }
941                 }
942
943                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
944                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
945
946                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
947
948                 // Can be worse than balanced variant
949                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
950
951                 //compressor1.ValidateFrequencies();
952             }
953         }
954
955         /// <summary>
956         /// <para>
957         /// Tests that all tree break down at sequences creation bug test.
958         /// </para>
959         /// <para></para>
960         /// </summary>
961         [Fact]
962         public static void AllTreeBreakDownAtSequencesCreationBugTest()
963         {
964             // Made out of AllPossibleConnectionsTest test.
965
966             //const long sequenceLength = 5; //100% bug
967             const long sequenceLength = 4; //100% bug
968             //const long sequenceLength = 3; //100% _no_bug (ok)
969
970             using (var scope = new TempLinksTestScope(useSequences: true))
971             {
972                 var links = scope.Links;
973                 var sequences = scope.Sequences;
974
975                 var sequence = new ulong[sequenceLength];
976                 for (var i = 0; i < sequenceLength; i++)
977                 {
978                     sequence[i] = links.Create();
979                 }
980
981                 var createResults = sequences.CreateAllVariants2(sequence);
982
983                 Global.Trash = createResults;
984
985                 for (var i = 0; i < sequenceLength; i++)
986                 {
987                     links.Delete(sequence[i]);
988                 }
989             }
990         }
991
992         /// <summary>
993         /// <para>
994         /// Tests that all possible connections test.
995         /// </para>
996         /// <para></para>
```

```
        /// </summary>
        [Fact]
        public static void AllPossibleConnectionsTest()
        {
            const long sequenceLength = 5;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
                }

                var createResults = sequences.CreateAllVariants2(sequence);
                var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var sw1 = Stopwatch.StartNew();
                    var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();

                    var sw2 = Stopwatch.StartNew();
                    var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();

                    var sw3 = Stopwatch.StartNew();
                    var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();

                    var sw4 = Stopwatch.StartNew();
                    var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();

                    Global.Trash = searchResults3;
                    Global.Trash = searchResults4; //-V3008

                    var intersection1 = createResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection1.Count == createResults.Length);

                    var intersection2 = reverseResults.Intersect(searchResults1).ToList();
                    Assert.True(intersection2.Count == reverseResults.Length);

                    var intersection0 = searchResults1.Intersect(searchResults2).ToList();
                    Assert.True(intersection0.Count == searchResults2.Count);

                    var intersection3 = searchResults2.Intersect(searchResults3).ToList();
                    Assert.True(intersection3.Count == searchResults3.Count);

                    var intersection4 = searchResults3.Intersect(searchResults4).ToList();
                    Assert.True(intersection4.Count == searchResults4.Count);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Tests that calculate all usages test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "Correct implementation is pending")]
        public static void CalculateAllUsagesTest()
        {
            const long sequenceLength = 3;

            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

                var sequence = new ulong[sequenceLength];
                for (var i = 0; i < sequenceLength; i++)
                {
                    sequence[i] = links.Create();
```

```
                }

                var createResults = sequences.CreateAllVariants2(sequence);

                //var reverseResults =
                ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());

                for (var i = 0; i < 1; i++)
                {
                    var linksTotalUsages1 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages(linksTotalUsages1);

                    var linksTotalUsages2 = new ulong[links.Count() + 1];

                    sequences.CalculateAllUsages2(linksTotalUsages2);

                    var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
                    Assert.True(intersection1.Count == linksTotalUsages2.Length);
                }

                for (var i = 0; i < sequenceLength; i++)
                {
                    links.Delete(sequence[i]);
                }
            }
        }
    }
}
```

## 1.68 ./csharp/Platform.Data.Doublets.Sequences.Tests/TempLinksTestScope.cs

```
using System.IO;
using Platform.Disposables;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;
using Platform.Data.Doublets.Memory.Split.Specific;
using Platform.Memory;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the temp links test scope.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DisposableBase"/>
    public class TempLinksTestScope : DisposableBase
    {
        /// <summary>
        /// <para>
        /// Gets the memory adapter value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ILinks<ulong> MemoryAdapter { get; }
        /// <summary>
        /// <para>
        /// Gets the links value.
        /// </para>
        /// <para></para>
        /// </summary>
        public SynchronizedLinks<ulong> Links { get; }
        /// <summary>
        /// <para>
        /// Gets the sequences value.
        /// </para>
        /// <para></para>
        /// </summary>
        public Sequences Sequences { get; }
        /// <summary>
        /// <para>
        /// Gets the temp filename value.
        /// </para>
        /// <para></para>
        /// </summary>
        public string TempFilename { get; }
        /// <summary>
```

```
        /// <para>
        /// Gets the temp transaction log filename value.
        /// </para>
        /// <para></para>
        /// </summary>
        public string TempTransactionLogFilename { get; }
        /// <summary>
        /// <para>
        /// The delete files.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly bool _deleteFiles;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TempLinksTestScope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="deleteFiles">
        /// <para>A delete files.</para>
        /// <para></para>
        /// </param>
        /// <param name="useSequences">
        /// <para>A use sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="useLog">
        /// <para>A use log.</para>
        /// <para></para>
        /// </param>
        public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
        ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
        ↪  useLog) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TempLinksTestScope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequencesOptions">
        /// <para>A sequences options.</para>
        /// <para></para>
        /// </param>
        /// <param name="deleteFiles">
        /// <para>A delete files.</para>
        /// <para></para>
        /// </param>
        /// <param name="useSequences">
        /// <para>A use sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="useLog">
        /// <para>A use log.</para>
        /// <para></para>
        /// </param>
        public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
        ↪  true, bool useSequences = false, bool useLog = false)
        {
            _deleteFiles = deleteFiles;
            TempFilename = Path.GetTempFileName();
            TempTransactionLogFilename = Path.GetTempFileName();
            //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
            var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
            ↪  FileMappedResizableDirectMemory(TempFilename), new
            ↪  FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
            ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
            ↪  Memory.IndexTreeType.Default, useLinkedList: true);
            MemoryAdapter = useLog ? (ILinks<ulong>)new
            ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
            ↪  coreMemoryAdapter;
            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
            if (useSequences)
            {
                Sequences = new Sequences(Links, sequencesOptions);
```

```
            }
        }

        /// <summary>
        /// <para>
        /// Disposes the manual.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="manual">
        /// <para>The manual.</para>
        /// <para></para>
        /// </param>
        /// <param name="wasDisposed">
        /// <para>The was disposed.</para>
        /// <para></para>
        /// </param>
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Links.Unsync.DisposeIfPossible();
                if (_deleteFiles)
                {
                    DeleteFiles();
                }
            }
        }

        /// <summary>
        /// <para>
        /// Deletes the files.
        /// </para>
        /// <para></para>
        /// </summary>
        public void DeleteFiles()
        {
            File.Delete(TempFilename);
            File.Delete(TempTransactionLogFilename);
        }
    }
}
```

## 1.69 ./csharp/Platform.Data.Doublets.Sequences.Tests/TestExtensions.cs

```
using System.Collections.Generic;
using Xunit;
using Platform.Ranges;
using Platform.Numbers;
using Platform.Random;
using Platform.Setters;
using Platform.Converters;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the test extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class TestExtensions
    {
        /// <summary>
        /// <para>
        /// Tests the crud operations using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        public static void TestCRUDOperations<T>(this ILinks<T> links)
        {
            var constants = links.Constants;
```

```
36
37              var equalityComparer = EqualityComparer<T>.Default;
38
39              var zero = default(T);
40              var one = Arithmetic.Increment(zero);
41
42              // Create Link
43              Assert.True(equalityComparer.Equals(links.Count(), zero));
44
45              var setter = new Setter<T>(constants.Null);
46              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
47
48              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
49
50              var linkAddress = links.Create();
51
52              var link = new Link<T>(links.GetLink(linkAddress));
53
54              Assert.True(link.Count == 3);
55              Assert.True(equalityComparer.Equals(link.Index, linkAddress));
56              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
57              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
58
59              Assert.True(equalityComparer.Equals(links.Count(), one));
60
61              // Get first link
62              setter = new Setter<T>(constants.Null);
63              links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
64
65              Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
66
67              // Update link to reference itself
68              links.Update(linkAddress, linkAddress, linkAddress);
69
70              link = new Link<T>(links.GetLink(linkAddress));
71
72              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
73              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
74
75              // Update link to reference null (prepare for delete)
76              var updated = links.Update(linkAddress, constants.Null, constants.Null);
77
78              Assert.True(equalityComparer.Equals(updated, linkAddress));
79
80              link = new Link<T>(links.GetLink(linkAddress));
81
82              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
83              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
84
85              // Delete link
86              links.Delete(linkAddress);
87
88              Assert.True(equalityComparer.Equals(links.Count(), zero));
89
90              setter = new Setter<T>(constants.Null);
91              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
92
93              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
94          }
95
96          /// <summary>
97          /// <para>
98          /// Tests the raw numbers crud operations using the specified links.
99          /// </para>
100         /// <para></para>
101         /// </summary>
102         /// <typeparam name="T">
103         /// <para>The .</para>
104         /// <para></para>
105         /// </typeparam>
106         /// <param name="links">
107         /// <para>The links.</para>
108         /// <para></para>
109         /// </param>
110         public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
111         {
112             // Constants
113             var constants = links.Constants;
114             var equalityComparer = EqualityComparer<T>.Default;
115
```

```
            var zero = default(T);
            var one = Arithmetic.Increment(zero);
            var two = Arithmetic.Increment(one);

            var h106E = new Hybrid<T>(106L, isExternal: true);
            var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
            var h108E = new Hybrid<T>(-108L);

            Assert.Equal(106L, h106E.AbsoluteValue);
            Assert.Equal(107L, h107E.AbsoluteValue);
            Assert.Equal(108L, h108E.AbsoluteValue);

            // Create Link (External -> External)
            var linkAddress1 = links.Create();

            links.Update(linkAddress1, h106E, h108E);

            var link1 = new Link<T>(links.GetLink(linkAddress1));

            Assert.True(equalityComparer.Equals(link1.Source, h106E));
            Assert.True(equalityComparer.Equals(link1.Target, h108E));

            // Create Link (Internal -> External)
            var linkAddress2 = links.Create();

            links.Update(linkAddress2, linkAddress1, h108E);

            var link2 = new Link<T>(links.GetLink(linkAddress2));

            Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link2.Target, h108E));

            // Create Link (Internal -> Internal)
            var linkAddress3 = links.Create();

            links.Update(linkAddress3, linkAddress1, linkAddress2);

            var link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
            Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));

            // Search for created link
            var setter1 = new Setter<T>(constants.Null);
            links.Each(h106E, h108E, setter1.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));

            // Search for nonexistent link
            var setter2 = new Setter<T>(constants.Null);
            links.Each(h106E, h107E, setter2.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress3, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress3));

            link3 = new Link<T>(links.GetLink(linkAddress3));

            Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link3.Target, constants.Null));

            // Delete link
            links.Delete(linkAddress3);

            Assert.True(equalityComparer.Equals(links.Count(), two));

            var setter3 = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
        }

        /// <summary>
        /// <para>
        /// Tests the multiple random creations and deletions using the specified links.
        /// </para>
        /// <para></para>
```

```
196             /// </summary>
197             /// <typeparam name="TLink">
198             /// <para>The link.</para>
199             /// <para></para>
200             /// </typeparam>
201             /// <param name="links">
202             /// <para>The links.</para>
203             /// <para></para>
204             /// </param>
205             /// <param name="maximumOperationsPerCycle">
206             /// <para>The maximum operations per cycle.</para>
207             /// <para></para>
208             /// </param>
209             public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
                ↪ links, int maximumOperationsPerCycle)
210             {
211                 var comparer = Comparer<TLink>.Default;
212                 var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
213                 var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
214                 for (var N = 1; N < maximumOperationsPerCycle; N++)
215                 {
216                     var random = new System.Random(N);
217                     var created = 0UL;
218                     var deleted = 0UL;
219                     for (var i = 0; i < N; i++)
220                     {
221                         var linksCount = addressToUInt64Converter.Convert(links.Count());
222                         var createPoint = random.NextBoolean();
223                         if (linksCount >= 2 && createPoint)
224                         {
225                             var linksAddressRange = new Range<ulong>(1, linksCount);
226                             TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                                ↪ ddressRange));
227                             TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                                ↪ ddressRange));
                                ↪ //-V3086
228                             var resultLink = links.GetOrCreate(source, target);
229                             if (comparer.Compare(resultLink,
                                ↪ uInt64ToAddressConverter.Convert(linksCount)) > 0)
230                             {
231                                 created++;
232                             }
233                         }
234                         else
235                         {
236                             links.Create();
237                             created++;
238                         }
239                     }
240                     Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
241                     for (var i = 0; i < N; i++)
242                     {
243                         TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
244                         if (links.Exists(link))
245                         {
246                             links.Delete(link);
247                             deleted++;
248                         }
249                     }
250                     Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
251                 }
252             }
253         }
254     }
```

## 1.70 ./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksTests.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Diagnostics;
4   using System.IO;
5   using System.Text;
6   using System.Threading;
7   using System.Threading.Tasks;
8   using Xunit;
9   using Platform.Disposables;
10  using Platform.Ranges;
11  using Platform.Random;
12  using Platform.Timestamps;
13  using Platform.Reflection;
```

```
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the int 64 links tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class UInt64LinksTests
    {
        /// <summary>
        /// <para>
        /// The instance.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        /// <summary>
        /// <para>
        /// The iterations.
        /// </para>
        /// <para></para>
        /// </summary>
        private const long Iterations = 10 * 1024;

        #region Concept

        /// <summary>
        /// <para>
        /// Tests that multiple create and delete test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void MultipleCreateAndDeleteTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UInt64UnitedMemoryLinks>>())
            {
                new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeleti⌋
                ↪  ons(100);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that cascade update test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CascadeUpdateTest()
        {
            var itself = _constants.Itself;
            using (var scope = new TempLinksTestScope(useLog: true))
            {
                var links = scope.Links;

                var l1 = links.Create();
                var l2 = links.Create();

                l2 = links.Update(l2, l2, l1, l2);

                links.CreateAndUpdate(l2, itself);
                links.CreateAndUpdate(l2, itself);

                l2 = links.Update(l2, l1);

```

```
90                  links.Delete(l2);
91
92                  Global.Trash = links.Count();
93
94                  links.Unsync.DisposeIfPossible(); // Close links to access log
95
96                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop⌋
                        ↪ e.TempTransactionLogFilename);
97              }
98          }
99
100         /// <summary>
101         /// <para>
102         /// Tests that basic transaction log test.
103         /// </para>
104         /// <para></para>
105         /// </summary>
106         [Fact]
107         public static void BasicTransactionLogTest()
108         {
109             using (var scope = new TempLinksTestScope(useLog: true))
110             {
111                 var links = scope.Links;
112                 var l1 = links.Create();
113                 var l2 = links.Create();
114
115                 Global.Trash = links.Update(l2, l2, l1, l2);
116
117                 links.Delete(l1);
118
119                 links.Unsync.DisposeIfPossible(); // Close links to access log
120
121                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop⌋
                        ↪ e.TempTransactionLogFilename);
122             }
123         }
124
125         /// <summary>
126         /// <para>
127         /// Tests that transaction auto reverted test.
128         /// </para>
129         /// <para></para>
130         /// </summary>
131         [Fact]
132         public static void TransactionAutoRevertedTest()
133         {
134             // Auto Reverted (Because no commit at transaction)
135             using (var scope = new TempLinksTestScope(useLog: true))
136             {
137                 var links = scope.Links;
138                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
139                 using (var transaction = transactionsLayer.BeginTransaction())
140                 {
141                     var l1 = links.Create();
142                     var l2 = links.Create();
143
144                     links.Update(l2, l2, l1, l2);
145                 }
146
147                 Assert.Equal(0UL, links.Count());
148
149                 links.Unsync.DisposeIfPossible();
150
151                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s⌋
                        ↪ cope.TempTransactionLogFilename);
152                 Assert.Single(transitions);
153             }
154         }
155
156         /// <summary>
157         /// <para>
158         /// Tests that transaction user code error no data saved test.
159         /// </para>
160         /// <para></para>
161         /// </summary>
162         [Fact]
163         public static void TransactionUserCodeErrorNoDataSavedTest()
164         {
165             // User Code Error (Autoreverted), no data saved
```

```
            var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪  useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                    ↪  atorBase<ulong>)links.Unsync).Links;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        var l1 = links.CreateAndUpdate(itself, itself);
                        var l2 = links.CreateAndUpdate(itself, itself);

                        l2 = links.Update(l2, l2, l1, l2);

                        links.CreateAndUpdate(l2, itself);
                        links.CreateAndUpdate(l2, itself);

                        //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                        ↪  tion>(scope.TempTransactionLogFilename);

                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Assert.False(lastScope == null);

                var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                ↪  astScope.TempTransactionLogFilename);

                Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                ↪  transitions[0].After.IsNull());

                lastScope.DeleteFiles();
            }
        }

        /// <summary>
        /// <para>
        /// Tests that transaction user code error some data saved test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void TransactionUserCodeErrorSomeDataSavedTest()
        {
            // User Code Error (Autoreverted), some data saved
            var itself = _constants.Itself;

            TempLinksTestScope lastScope = null;
            try
            {
                ulong l1;
                ulong l2;

                using (var scope = new TempLinksTestScope(useLog: true))
                {
                    var links = scope.Links;
                    l1 = links.CreateAndUpdate(itself, itself);
                    l2 = links.CreateAndUpdate(itself, itself);

                    l2 = links.Update(l2, l2, l1, l2);

                    links.CreateAndUpdate(l2, itself);
                    links.CreateAndUpdate(l2, itself);

                    links.Unsync.DisposeIfPossible();
```

```
241
242                     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                        ↪ scope.TempTransactionLogFilename);
243                 }
244
245                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪ useLog: true))
246                 {
247                     var links = scope.Links;
248                     var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
249                     using (var transaction = transactionsLayer.BeginTransaction())
250                     {
251                         l2 = links.Update(l2, l1);
252
253                         links.Delete(l2);
254
255                         ExceptionThrower();
256
257                         transaction.Commit();
258                     }
259
260                     Global.Trash = links.Count();
261                 }
262             }
263             catch
264             {
265                 Assert.False(lastScope == null);
266
267                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                    ↪ Scope.TempTransactionLogFilename);
268
269                 lastScope.DeleteFiles();
270             }
271         }
272
273         /// <summary>
274         /// <para>
275         /// Tests that transaction commit.
276         /// </para>
277         /// <para></para>
278         /// </summary>
279         [Fact]
280         public static void TransactionCommit()
281         {
282             var itself = _constants.Itself;
283
284             var tempDatabaseFilename = Path.GetTempFileName();
285             var tempTransactionLogFilename = Path.GetTempFileName();
286
287             // Commit
288             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
289             using (var links = new UInt64Links(memoryAdapter))
290             {
291                 using (var transaction = memoryAdapter.BeginTransaction())
292                 {
293                     var l1 = links.CreateAndUpdate(itself, itself);
294                     var l2 = links.CreateAndUpdate(itself, itself);
295
296                     Global.Trash = links.Update(l2, l2, l1, l2);
297
298                     links.Delete(l1);
299
300                     transaction.Commit();
301                 }
302
303                 Global.Trash = links.Count();
304             }
305
306             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                    ↪ sactionLogFilename);
307         }
308
309         /// <summary>
310         /// <para>
311         /// Tests that transaction damage.
312         /// </para>
313         /// <para></para>
314         /// </summary>
```

```
315         [Fact]
316         public static void TransactionDamage()
317         {
318             var itself = _constants.Itself;
319
320             var tempDatabaseFilename = Path.GetTempFileName();
321             var tempTransactionLogFilename = Path.GetTempFileName();
322
323             // Commit
324             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
325             using (var links = new UInt64Links(memoryAdapter))
326             {
327                 using (var transaction = memoryAdapter.BeginTransaction())
328                 {
329                     var l1 = links.CreateAndUpdate(itself, itself);
330                     var l2 = links.CreateAndUpdate(itself, itself);
331
332                     Global.Trash = links.Update(l2, l2, l1, l2);
333
334                     links.Delete(l1);
335
336                     transaction.Commit();
337                 }
338
339                 Global.Trash = links.Count();
340             }
341
342             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
343
344             // Damage database
345
346             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
347
348             // Try load damaged database
349             try
350             {
351                 // TODO: Fix
352                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
353                 using (var links = new UInt64Links(memoryAdapter))
354                 {
355                     Global.Trash = links.Count();
356                 }
357             }
358             catch (NotSupportedException ex)
359             {
360                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
361             }
362
363             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
364
365             File.Delete(tempDatabaseFilename);
366             File.Delete(tempTransactionLogFilename);
367         }
368
369         /// <summary>
370         /// <para>
371         /// Tests that bug 1 test.
372         /// </para>
373         /// <para></para>
374         /// </summary>
375         [Fact]
376         public static void Bug1Test()
377         {
378             var tempDatabaseFilename = Path.GetTempFileName();
379             var tempTransactionLogFilename = Path.GetTempFileName();
380
381             var itself = _constants.Itself;
382
383             // User Code Error (Autoreverted), some data saved
384             try
385             {
386                 ulong l1;
387                 ulong l2;
```

```
388
389                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
390                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪ tempTransactionLogFilename))
391                 using (var links = new UInt64Links(memoryAdapter))
392                 {
393                     l1 = links.CreateAndUpdate(itself, itself);
394                     l2 = links.CreateAndUpdate(itself, itself);
395
396                     l2 = links.Update(l2, l2, l1, l2);
397
398                     links.CreateAndUpdate(l2, itself);
399                     links.CreateAndUpdate(l2, itself);
400                 }
401
402                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
403
404                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
405                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                    ↪ tempTransactionLogFilename))
406                 using (var links = new UInt64Links(memoryAdapter))
407                 {
408                     using (var transaction = memoryAdapter.BeginTransaction())
409                     {
410                         l2 = links.Update(l2, l1);
411
412                         links.Delete(l2);
413
414                         ExceptionThrower();
415
416                         transaction.Commit();
417                     }
418
419                     Global.Trash = links.Count();
420                 }
421             }
422             catch
423             {
424                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                    ↪ TransactionLogFilename);
425             }
426
427             File.Delete(tempDatabaseFilename);
428             File.Delete(tempTransactionLogFilename);
429         }
430
431         /// <summary>
432         /// <para>
433         /// Exceptions the thrower.
434         /// </para>
435         /// <para></para>
436         /// </summary>
437         private static void ExceptionThrower() => throw new InvalidOperationException();
438
439         /// <summary>
440         /// <para>
441         /// Tests that paths test.
442         /// </para>
443         /// <para></para>
444         /// </summary>
445         [Fact]
446         public static void PathsTest()
447         {
448             var source = _constants.SourcePart;
449             var target = _constants.TargetPart;
450
451             using (var scope = new TempLinksTestScope())
452             {
453                 var links = scope.Links;
454                 var l1 = links.CreatePoint();
455                 var l2 = links.CreatePoint();
456
457                 var r1 = links.GetByKeys(l1, source, target, source);
458                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
459             }
460         }
461
462         /// <summary>
```

```
        /// <para>
        /// Tests that recursive string formatting test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void RecursiveStringFormattingTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.

                var a = links.CreatePoint();
                var b = links.CreatePoint();
                var c = links.CreatePoint();

                var ab = links.GetOrCreate(a, b);
                var cb = links.GetOrCreate(c, b);
                var ac = links.GetOrCreate(a, c);

                a = links.Update(a, c, b);
                b = links.Update(b, a, c);
                c = links.Update(c, a, b);

                Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
                Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

                Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                ↪  "(5:(4:5 (6:5 4)) 6)");
                Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                ↪  "(6:(5:(4:5 6) 6) 4)");
                Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                ↪  "(4:(5:4 (6:5 4)) 6)");

                // TODO: Think how to build balanced syntax tree while formatting structure (eg.
                ↪  "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"

                Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                ↪  "{{5}{5}{4}{6}}");
                Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                ↪  "{{5}{6}{6}{4}}");
                Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                ↪  "{{4}{5}{4}{6}}");
            }
        }

        /// <summary>
        /// <para>
        /// Defaults the formatter using the specified sb.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sb">
        /// <para>The sb.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        private static void DefaultFormatter(StringBuilder sb, ulong link)
        {
            sb.Append(link.ToString());
        }

        #endregion

        #region Performance

        /*
        public static void RunAllPerformanceTests()
        {
            try
            {
                links.TestLinksInSteps();
            }
```

```
534             catch (Exception ex)
535             {
536                 ex.WriteToConsole();
537             }
538 
539             return;
540 
541             try
542             {
543                 //ThreadPool.SetMaxThreads(2, 2);
544 
545                 // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
546                 // Также это дополнительно помогает в отладке
547                 // Увеличивает вероятность попадания информации в кэши
548                 for (var i = 0; i < 10; i++)
549                 {
550                     //0 - 10 ГБ
551                     //Каждые 100 МБ срез цифр
552 
553                     //links.TestGetSourceFunction();
554                     //links.TestGetSourceFunctionInParallel();
555                     //links.TestGetTargetFunction();
556                     //links.TestGetTargetFunctionInParallel();
557                     links.Create64BillionLinks();
558 
559                     links.TestRandomSearchFixed();
560                     //links.Create64BillionLinksInParallel();
561                     links.TestEachFunction();
562                     //links.TestForeach();
563                     //links.TestParallelForeach();
564                 }
565 
566                 links.TestDeletionOfAllLinks();
567 
568             }
569             catch (Exception ex)
570             {
571                 ex.WriteToConsole();
572             }
573         }*/
574 
575          /*
576         public static void TestLinksInSteps()
577         {
578             const long gibibyte = 1024 * 1024 * 1024;
579             const long mebibyte = 1024 * 1024;
580 
581             var totalLinksToCreate = gibibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
582             var linksStep = 102 * mebibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
583 
584             var creationMeasurements = new List<TimeSpan>();
585             var searchMeasuremets = new List<TimeSpan>();
586             var deletionMeasurements = new List<TimeSpan>();
587 
588             GetBaseRandomLoopOverhead(linksStep);
589             GetBaseRandomLoopOverhead(linksStep);
590 
591             var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
592 
593             ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
594 
595             var loops = totalLinksToCreate / linksStep;
596 
597             for (int i = 0; i < loops; i++)
598             {
599                 creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
600                 searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
601 
602                 Console.Write("\rC + S {0}/{1}", i + 1, loops);
603             }
604 
605             ConsoleHelpers.Debug();
606 
607             for (int i = 0; i < loops; i++)
608             {
609                 deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
610 
```

```
611                 Console.Write("\rD {0}/{1}", i + 1, loops);
612             }
613 
614             ConsoleHelpers.Debug();
615 
616             ConsoleHelpers.Debug("C S D");
617 
618             for (int i = 0; i < loops; i++)
619             {
620                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
621             }
622 
623             ConsoleHelpers.Debug("C S D (no overhead)");
624 
625             for (int i = 0; i < loops; i++)
626             {
627                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
628             }
629 
630             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
631         }
632 
633         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
634         {
635             for (long i = 0; i < amountToCreate; i++)
636                 links.Create(0, 0);
637         }
638 
639          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
640          {
641              return Measure(() =>
642              {
643                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
644                  ulong result = 0;
645                  for (long i = 0; i < loops; i++)
646                  {
647                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
648                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
649 
650                      result += maxValue + source + target;
651                  }
652                  Global.Trash = result;
653              });
654          }
655           */
656 
657          /// <summary>
658          /// <para>
659          /// Tests that get source test.
660          /// </para>
661          /// <para></para>
662          /// </summary>
663          [Fact(Skip = "performance test")]
664          public static void GetSourceTest()
665          {
666              using (var scope = new TempLinksTestScope())
667              {
668                  var links = scope.Links;
669                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
670 
671                  ulong counter = 0;
672 
673                  //var firstLink = links.First();
674                  // Создаём одну связь, из которой будет производить считывание
675                  var firstLink = links.Create();
676 
677                  var sw = Stopwatch.StartNew();
678 
679                  // Тестируем саму функцию
680                  for (ulong i = 0; i < Iterations; i++)
681                  {
682                      counter += links.GetSource(firstLink);
683                  }
684 
```

```
685                 var elapsedTime = sw.Elapsed;
686
687                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
688
689                 // Удаляем связь, из которой производилось считывание
690                 links.Delete(firstLink);
691
692                 ConsoleHelpers.Debug(
693                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
694                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
695             }
696         }
697
698         /// <summary>
699         /// <para>
700         /// Tests that get source in parallel.
701         /// </para>
702         /// <para></para>
703         /// </summary>
704         [Fact(Skip = "performance test")]
705         public static void GetSourceInParallel()
706         {
707             using (var scope = new TempLinksTestScope())
708             {
709                 var links = scope.Links;
710                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
711
712                 long counter = 0;
713
714                 //var firstLink = links.First();
715                 var firstLink = links.Create();
716
717                 var sw = Stopwatch.StartNew();
718
719                 // Тестируем саму функцию
720                 Parallel.For(0, Iterations, x =>
721                 {
722                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
723                     //Interlocked.Increment(ref counter);
724                 });
725
726                 var elapsedTime = sw.Elapsed;
727
728                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
729
730                 links.Delete(firstLink);
731
732                 ConsoleHelpers.Debug(
733                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
734                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
735             }
736         }
737
738         /// <summary>
739         /// <para>
740         /// Tests that test get target.
741         /// </para>
742         /// <para></para>
743         /// </summary>
744         [Fact(Skip = "performance test")]
745         public static void TestGetTarget()
746         {
747             using (var scope = new TempLinksTestScope())
748             {
749                 var links = scope.Links;
750                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
751
752                 ulong counter = 0;
753
754                 //var firstLink = links.First();
755                 var firstLink = links.Create();
756
757                 var sw = Stopwatch.StartNew();
758
759                 for (ulong i = 0; i < Iterations; i++)
```

```
760                 {
761                     counter += links.GetTarget(firstLink);
762                 }
763 
764                 var elapsedTime = sw.Elapsed;
765 
766                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
767 
768                 links.Delete(firstLink);
769 
770                 ConsoleHelpers.Debug(
771                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
772                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
773             }
774         }
775 
776         /// <summary>
777         /// <para>
778         /// Tests that test get target in parallel.
779         /// </para>
780         /// <para></para>
781         /// </summary>
782         [Fact(Skip = "performance test")]
783         public static void TestGetTargetInParallel()
784         {
785             using (var scope = new TempLinksTestScope())
786             {
787                 var links = scope.Links;
788                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                    ↪  parallel.", Iterations);
789 
790                 long counter = 0;
791 
792                 //var firstLink = links.First();
793                 var firstLink = links.Create();
794 
795                 var sw = Stopwatch.StartNew();
796 
797                 Parallel.For(0, Iterations, x =>
798                 {
799                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
800                     //Interlocked.Increment(ref counter);
801                 });
802 
803                 var elapsedTime = sw.Elapsed;
804 
805                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
806 
807                 links.Delete(firstLink);
808 
809                 ConsoleHelpers.Debug(
810                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
811                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
812             }
813         }
814 
815         // TODO: Заполнить базу данных перед тестом
816         /*
817         [Fact]
818         public void TestRandomSearchFixed()
819         {
820             var tempFilename = Path.GetTempFileName();
821 
822             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
823             {
824                 long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
825 
826                 ulong counter = 0;
827                 var maxLink = links.Total;
828 
829                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
830 
831                 var sw = Stopwatch.StartNew();
832 
833                 for (var i = iterations; i > 0; i--)
```

```
834                 {
835                     var source =
    ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
836                     var target =
    ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
837 
838                     counter += links.Search(source, target);
839                 }
840 
841                 var elapsedTime = sw.Elapsed;
842 
843                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
844 
845                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
    ↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
    ↪ counter);
846             }
847 
848             File.Delete(tempFilename);
849         }*/
850 
851         /// <summary>
852         /// <para>
853         /// Tests that test random search all.
854         /// </para>
855         /// <para></para>
856         /// </summary>
857         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
858         public static void TestRandomSearchAll()
859         {
860             using (var scope = new TempLinksTestScope())
861             {
862                 var links = scope.Links;
863                 ulong counter = 0;
864 
865                 var maxLink = links.Count();
866 
867                 var iterations = links.Count();
868 
869                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                    ↪ links.Count());
870 
871                 var sw = Stopwatch.StartNew();
872 
873                 for (var i = iterations; i > 0; i--)
874                 {
875                     var linksAddressRange = new
                        ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
876 
877                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
878                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
879 
880                     counter += links.SearchOrDefault(source, target);
881                 }
882 
883                 var elapsedTime = sw.Elapsed;
884 
885                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
886 
887                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                    ↪ Iterations per second), c: {3}",
888                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
889             }
890         }
891 
892         /// <summary>
893         /// <para>
894         /// Tests that test each.
895         /// </para>
896         /// <para></para>
897         /// </summary>
898         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
899         public static void TestEach()
900         {
901             using (var scope = new TempLinksTestScope())
902             {
903                 var links = scope.Links;
904 
905                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
```

```
906
907                 ConsoleHelpers.Debug("Testing Each function.");
908
909                 var sw = Stopwatch.StartNew();
910
911                 links.Each(counter.IncrementAndReturnTrue);
912
913                 var elapsedTime = sw.Elapsed;
914
915                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
916
917                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
918                     counter, elapsedTime, (long)linksPerSecond);
919             }
920         }
921
922         /*
923         [Fact]
924         public static void TestForeach()
925         {
926             var tempFilename = Path.GetTempFileName();
927
928             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
929             {
930                 ulong counter = 0;
931
932                 ConsoleHelpers.Debug("Testing foreach through links.");
933
934                 var sw = Stopwatch.StartNew();
935
936                 //foreach (var link in links)
937                 //{
938                 //    counter++;
939                 //}
940
941                 var elapsedTime = sw.Elapsed;
942
943                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
944
945                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
946             }
947
948             File.Delete(tempFilename);
949         }
950         */
951
952         /*
953         [Fact]
954         public static void TestParallelForeach()
955         {
956             var tempFilename = Path.GetTempFileName();
957
958             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
959             {
960
961                 long counter = 0;
962
963                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
964
965                 var sw = Stopwatch.StartNew();
966
967                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
968                 //{
969                 //    Interlocked.Increment(ref counter);
970                 //});
971
972                 var elapsedTime = sw.Elapsed;
973
974                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
975
976                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
977             }
978
979             File.Delete(tempFilename);
980         }
```

```
        */

        /// <summary>
        /// <para>
        /// Tests that create 64 billion links.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "performance test")]
        public static void Create64BillionLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

                var elapsedTime = Performance.Measure(() =>
                {
                    for (long i = 0; i < linksToCreate; i++)
                    {
                        links.Create();
                    }
                });

                var linksCreated = links.Count() - linksBeforeTest;
                var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

                ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                ↪  linksCreated, elapsedTime,
                    (long)linksPerSecond);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that create 64 billion links in parallel.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "performance test")]
        public static void Create64BillionLinksInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                var sw = Stopwatch.StartNew();

                long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

                ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);

                Parallel.For(0, linksToCreate, x => links.Create());

                var elapsedTime = sw.Elapsed;

                var linksCreated = links.Count() - linksBeforeTest;
                var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                ↪  linksCreated, elapsedTime,
                    (long)linksPerSecond);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that test deletion of all links.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
```

```
        public static void TestDeletionOfAllLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                ConsoleHelpers.Debug("Deleting all links");

                var elapsedTime = Performance.Measure(links.DeleteAll);

                var linksDeleted = linksBeforeTest - links.Count();
                var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
                    linksDeleted, elapsedTime,
                    (long)linksPerSecond);
            }
        }

        #endregion
    }
}
```

## 1.71 ./csharp/Platform.Data.Doublets.Sequences.Tests/Uint64LinksExtensionsTests.cs

```
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the uint 64 links extensions tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class Uint64LinksExtensionsTests
    {
        /// <summary>
        /// <para>
        /// Creates the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
            Platform.IO.TemporaryFile());

        /// <summary>
        /// <para>
        /// Creates the links using the specified data db filename.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dataDBFilename">
        /// <para>The data db filename.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A links of t link</para>
        /// <para></para>
        /// </returns>
        public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
        {
            var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                true);
```

```
53              return new UnitedMemoryLinks<TLink>(new
                ↪  FileMappedResizableDirectMemory(dataDBFilename),
                ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
54          }
55          /// <summary>
56          /// <para>
57          /// Tests that format structure with external reference test.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          [Fact]
62          public void FormatStructureWithExternalReferenceTest()
63          {
64              ILinks<TLink> links = CreateLinks();
65              TLink zero = default;
66              var one = Arithmetic.Increment(zero);
67              var markerIndex = one;
68              var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
69              var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
                ↪  markerIndex));
70              AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
71              var numberAddress = addressToNumberConverter.Convert(1);
72              var numberLink = links.GetOrCreate(numberMarker, numberAddress);
73              var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
                ↪  true);
74              Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
75          }
76      }
77  }
```

## 1.72 ./csharp/Platform.Data.Doublets.Sequences.Tests/UnaryNumberConvertersTests.cs

```
1   using Xunit;
2   using Platform.Random;
3   using Platform.Data.Doublets.Numbers.Unary;
4
5   namespace Platform.Data.Doublets.Sequences.Tests
6   {
7       /// <summary>
8       /// <para>
9       /// Represents the unary number converters tests.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      public static class UnaryNumberConvertersTests
14      {
15          /// <summary>
16          /// <para>
17          /// Tests that converters test.
18          /// </para>
19          /// <para></para>
20          /// </summary>
21          [Fact]
22          public static void ConvertersTest()
23          {
24              using (var scope = new TempLinksTestScope())
25              {
26                  const int N = 10;
27                  var links = scope.Links;
28                  var meaningRoot = links.CreatePoint();
29                  var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
30                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
31                  var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
32                  var random = new System.Random(0);
33                  ulong[] numbers = new ulong[N];
34                  ulong[] unaryNumbers = new ulong[N];
35                  for (int i = 0; i < N; i++)
36                  {
37                      numbers[i] = random.NextUInt64();
38                      unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
39                  }
40                  var fromUnaryNumberConverterUsingOrOperation = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
41                  var fromUnaryNumberConverterUsingAddOperation = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
```

```
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

## 1.73 ./csharp/Platform.Data.Doublets.Sequences.Tests/UnicodeConvertersTests.cs

```
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.CriterionMatchers;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the unicode converters tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class UnicodeConvertersTests
    {
        /// <summary>
        /// <para>
        /// Tests that char and unary number unicode symbol converters test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var addressToUnaryNumberConverter = new
                ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                ↪  addressToUnaryNumberConverter, unaryNumberToAddressConverter);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that char and raw number unicode symbol converters test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CharAndRawNumberUnicodeSymbolConvertersTest()
        {
            using (var scope = new Scope<Types<HeapResizableDirectMemory,
            ↪  UnitedMemoryLinks<ulong>>>())
            {
                var links = scope.Use<ILinks<ulong>>();
                var meaningRoot = links.CreatePoint();
```

```
61                  var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
62                  var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
63                  TestCharAndUnicodeSymbolConverters(links, meaningRoot,
                        ↪  addressToRawNumberConverter, rawNumberToAddressConverter);
64              }
65          }
66
67          /// <summary>
68          /// <para>
69          /// Tests the char and unicode symbol converters using the specified links.
70          /// </para>
71          /// <para></para>
72          /// </summary>
73          /// <param name="links">
74          /// <para>The links.</para>
75          /// <para></para>
76          /// </param>
77          /// <param name="meaningRoot">
78          /// <para>The meaning root.</para>
79          /// <para></para>
80          /// </param>
81          /// <param name="addressToNumberConverter">
82          /// <para>The address to number converter.</para>
83          /// <para></para>
84          /// </param>
85          /// <param name="numberToAddressConverter">
86          /// <para>The number to address converter.</para>
87          /// <para></para>
88          /// </param>
89          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
            ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
            ↪  numberToAddressConverter)
90          {
91              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
92              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
                    ↪  addressToNumberConverter, unicodeSymbolMarker);
93              var originalCharacter = 'H';
94              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
95              var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSymbolMarker);
96              var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
                    ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
97              var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
98              Assert.Equal(originalCharacter, resultingCharacter);
99          }
100
101         /// <summary>
102         /// <para>
103         /// Tests that string and unicode sequence converters test.
104         /// </para>
105         /// <para></para>
106         /// </summary>
107         [Fact]
108         public static void StringAndUnicodeSequenceConvertersTest()
109         {
110             using (var scope = new TempLinksTestScope())
111             {
112                 var links = scope.Links;
113
114                 var itself = links.Constants.Itself;
115
116                 var meaningRoot = links.CreatePoint();
117                 var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
118                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
119                 var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
120                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
121                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
122
123                 var powerOf2ToUnaryNumberConverter = new
                        ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
124                 var addressToUnaryNumberConverter = new
                        ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
125                 var charToUnicodeSymbolConverter = new
                        ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                        ↪  unicodeSymbolMarker);
126
```

```
                var unaryNumberToAddressConverter = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
                var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
                var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                ↪  frequencyPropertyMarker, frequencyMarker);
                var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                ↪  frequencyPropertyOperator, frequencyIncrementer);
                var linkToItsFrequencyNumberConverter = new
                ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                ↪  unaryNumberToAddressConverter);
                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                ↪  sequenceToItsLocalElementLevelsConverter);

                var stringToUnicodeSequenceConverter = new
                ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                ↪  index, optimalVariantConverter, unicodeSequenceMarker);

                var originalString = "Hello";

                var unicodeSequenceLink =
                ↪  stringToUnicodeSequenceConverter.Convert(originalString);

                var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSymbolMarker);
                var unicodeSymbolToCharConverter = new
                ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                ↪  unicodeSymbolCriterionMatcher);

                var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSequenceMarker);

                var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                ↪  unicodeSymbolCriterionMatcher.IsMatched);

                var unicodeSequenceToStringConverter = new
                ↪  UnicodeSequenceToStringConverter<ulong>(links,
                ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                ↪  unicodeSymbolToCharConverter);

                var resultingString =
                ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);

                Assert.Equal(originalString, resultingString);
            }
        }
    }
}
```

Index