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>
```

```
        /// Halves the sequence using the specified destination.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="destination">
        /// <para>The destination.</para>
        /// <para></para>
        /// </param>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
        {
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Converters;
using Platform.Singletons;
using Platform.Numbers;
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.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на этапе сжатия.
    ///     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.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksConstants<TLink> _constants = Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

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

        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;

        private struct HalfDoublet
        {
            /// <summary>
            /// <para>
            /// The element.
            /// </para>
            /// <para></para>
            /// </summary>
```

```
43              public TLink Element;
44              /// <summary>
45              /// <para>
46              /// The doublet data.
47              /// </para>
48              /// <para></para>
49              /// </summary>
50              public LinkFrequency<TLink> DoubletData;
51
52              /// <summary>
53              /// <para>
54              /// Initializes a new <see cref="HalfDoublet"/> instance.
55              /// </para>
56              /// <para></para>
57              /// </summary>
58              /// <param name="element">
59              /// <para>A element.</para>
60              /// <para></para>
61              /// </param>
62              /// <param name="doubletData">
63              /// <para>A doublet data.</para>
64              /// <para></para>
65              /// </param>
66              [MethodImpl(MethodImplOptions.AggressiveInlining)]
67              public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
68              {
69                  Element = element;
70                  DoubletData = doubletData;
71              }
72
73              /// <summary>
74              /// <para>
75              /// Returns the string.
76              /// </para>
77              /// <para></para>
78              /// </summary>
79              /// <returns>
80              /// <para>The string</para>
81              /// <para></para>
82              /// </returns>
83              public override string ToString() => $"{Element}: ({DoubletData})";
84          }
85
86          /// <summary>
87          /// <para>
88          /// Initializes a new <see cref="CompressingConverter"/> instance.
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          /// <param name="links">
93          /// <para>A links.</para>
94          /// <para></para>
95          /// </param>
96          /// <param name="baseConverter">
97          /// <para>A base converter.</para>
98          /// <para></para>
99          /// </param>
100         /// <param name="doubletFrequenciesCache">
101         /// <para>A doublet frequencies cache.</para>
102         /// <para></para>
103         /// </param>
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
106             : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
107
108         /// <summary>
109         /// <para>
110         /// Initializes a new <see cref="CompressingConverter"/> instance.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="links">
115         /// <para>A links.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="baseConverter">
119         /// <para>A base converter.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="doubletFrequenciesCache">
        /// <para>A doublet frequencies cache.</para>
        /// <para></para>
        /// </param>
        /// <param name="doInitialFrequenciesIncrement">
        /// <para>A do initial frequencies increment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
        ↪  doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, _one,
            ↪  doInitialFrequenciesIncrement) { }

        /// <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>
        /// <param name="minFrequencyToCompress">
        /// <para>A min frequency to compress.</para>
        /// <para></para>
        /// </param>
        /// <param name="doInitialFrequenciesIncrement">
        /// <para>A do initial frequencies increment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
        ↪  baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
        ↪  minFrequencyToCompress, bool doInitialFrequenciesIncrement)
            : base(links)
        {
            _baseConverter = baseConverter;
            _doubletFrequenciesCache = doubletFrequenciesCache;
            if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
            {
                minFrequencyToCompress = _one;
            }
            _minFrequencyToCompress = minFrequencyToCompress;
            _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
            ResetMaxDoublet();
        }

        /// <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 override TLink Convert(IList<TLink> source) =>
        ↪  _baseConverter.Convert(Compress(source));

```

```
        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding .
        /// Faster version (doublets' frequencies dictionary is not recreated).
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private IList<TLink> Compress(IList<TLink> sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return null;
            }
            if (sequence.Count == 1)
            {
                return sequence;
            }
            if (sequence.Count == 2)
            {
                return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
            }
            // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
            var copy = new HalfDoublet[sequence.Count];
            Doublet<TLink> doublet = default;
            for (var i = 1; i < sequence.Count; i++)
            {
                doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
                LinkFrequency<TLink> data;
                if (_doInitialFrequenciesIncrement)
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                    if (data == null)
                    {
                        throw new NotSupportedException("If you ask not to increment
                        ↪  frequencies, it is expected that all frequencies for the sequence
                        ↪  are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }

        /// <remarks>
        /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte_pair_encoding
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                    ↪  maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
```

```
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    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);
            }
```

```
341         }
342
343         /// <summary>
344         /// <para>
345         /// Updates the max doublet using the specified doublet.
346         /// </para>
347         /// <para></para>
348         /// </summary>
349         /// <param name="doublet">
350         /// <para>The doublet.</para>
351         /// <para></para>
352         /// </param>
353         /// <param name="data">
354         /// <para>The data.</para>
355         /// <para></para>
356         /// </param>
357         [MethodImpl(MethodImplOptions.AggressiveInlining)]
358         private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
359         {
360             var frequency = data.Frequency;
361             var maxFrequency = _maxDoubletData.Frequency;
362             //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                ↪  _maxDoublet.Target)))
363             if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
364                 (_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) */
365             {
366                 _maxDoublet = doublet;
367                 _maxDoubletData = data;
368             }
369         }
370     }
371 }
```

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>
```

```
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        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)
```

```
60              : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen
                ↪  ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) {
                ↪  }
61
62          /// <summary>
63          /// <para>
64          /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          /// <param name="links">
69          /// <para>A links.</para>
70          /// <para></para>
71          /// </param>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public OptimalVariantConverter(ILinks<TLink> links)
74              : this(links, new LinkFrequenciesCache<TLink>(links, new
                ↪  TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
75
76          /// <summary>
77          /// <para>
78          /// Converts the sequence.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="sequence">
83          /// <para>The sequence.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public override TLink Convert(IList<TLink> sequence)
92          {
93              var length = sequence.Count;
94              if (length == 1)
95              {
96                  return sequence[0];
97              }
98              if (length == 2)
99              {
100                 return _links.GetOrCreate(sequence[0], sequence[1]);
101             }
102             sequence = sequence.ToArray();
103             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
104             while (length > 2)
105             {
106                 var levelRepeat = 1;
107                 var currentLevel = levels[0];
108                 var previousLevel = levels[0];
109                 var skipOnce = false;
110                 var w = 0;
111                 for (var i = 1; i < length; i++)
112                 {
113                     if (_equalityComparer.Equals(currentLevel, levels[i]))
114                     {
115                         levelRepeat++;
116                         skipOnce = false;
117                         if (levelRepeat == 2)
118                         {
119                             sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
120                             var newLevel = i >= length - 1 ?
121                                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                    ↪  currentLevel) :
122                                 i < 2 ?
123                                 GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
124                                 GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                    ↪  currentLevel, levels[i + 1]);
125                             levels[w] = newLevel;
126                             previousLevel = currentLevel;
127                             w++;
128                             levelRepeat = 0;
129                             skipOnce = true;
130                         }
131                         else if (i == length - 1)
132                         {
```

```
                        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>>
    {
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

        private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceToItsLocalElementLevelsConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkToItsFrequencyToNumberConveter">
        /// <para>A link to its frequency to number conveter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
        ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
        ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;

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

```
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <param name="sequence">
47          /// <para>The sequence.</para>
48          /// <para></para>
49          /// </param>
50          /// <returns>
51          /// <para>The levels.</para>
52          /// <para></para>
53          /// </returns>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public IList<TLink> Convert(IList<TLink> sequence)
56          {
57              var levels = new TLink[sequence.Count];
58              levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
59              for (var i = 1; i < sequence.Count - 1; i++)
60              {
61                  var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
62                  var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
63                  levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
64              }
65              levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
                ↪  sequence[sequence.Count - 1]);
66              return levels;
67          }
68
69          /// <summary>
70          /// <para>
71          /// Gets the frequency number using the specified source.
72          /// </para>
73          /// <para></para>
74          /// </summary>
75          /// <param name="source">
76          /// <para>The source.</para>
77          /// <para></para>
78          /// </param>
79          /// <param name="target">
80          /// <para>The target.</para>
81          /// <para></para>
82          /// </param>
83          /// <returns>
84          /// <para>The link</para>
85          /// <para></para>
86          /// </returns>
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public TLink GetFrequencyNumber(TLink source, TLink target) =>
            ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
89      }
90  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.CriterionMatchers
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the default sequence element criterion matcher.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="LinksOperatorBase{TLink}"/>
15      /// <seealso cref="ICriterionMatcher{TLink}"/>
16      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↪  ICriterionMatcher<TLink>
17      {
18          /// <summary>
19          /// <para>
20          /// Initializes a new <see cref="DefaultSequenceElementCriterionMatcher"/> instance.
21          /// </para>
22          /// <para></para>
23          /// </summary>
24          /// <param name="links">
25          /// <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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly ILinks<TLink> _links;
        private readonly TLink _sequenceMarkerLink;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="MarkedSequenceCriterionMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceMarkerLink">
        /// <para>A sequence marker link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceCandidate">
        /// <para>The sequence candidate.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

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

        private readonly IStack<TLink> _stack;
        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>
    {
        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.
```

```
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          /// <returns>
41          /// <para>The int</para>
42          /// <para></para>
43          /// </returns>
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
46      }
47  }
```

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

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Interfaces;
6   using Platform.Collections;
7   using Platform.Collections.Lists;
8   using Platform.Collections.Segments;
9   using Platform.Collections.Segments.Walkers;
10  using Platform.Singletons;
11  using Platform.Converters;
12  using Platform.Data.Doublets.Unicode;
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 duplicate segments provider.
21      /// </para>
22      /// <para></para>
23      /// </summary>
24      /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLink}"/>
25      /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLink}, IList{TLink}}}}"/>
26      public class DuplicateSegmentsProvider<TLink> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
27      {
28          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
29          private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            ↪  UncheckedConverter<TLink, ulong>.Default;
30          private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            ↪  UncheckedConverter<ulong, TLink>.Default;
31
32          private readonly ILinks<TLink> _links;
33          private readonly ILinks<TLink> _sequences;
34          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
35          private BitString _visited;
36
37          private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>
38          {
39              private readonly IListEqualityComparer<TLink> _listComparer;
40
41              /// <summary>
42              /// <para>
43              /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
44              /// </para>
45              /// <para></para>
46              /// </summary>
47              public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
48
49              /// <summary>
50              /// <para>
51              /// Determines whether this instance equals.
52              /// </para>
53              /// <para></para>
54              /// </summary>
55              /// <param name="left">
56              /// <para>The left.</para>
57              /// <para></para>
58              /// </param>
59              /// <param name="right">
60              /// <para>The right.</para>
```

```
61              /// <para></para>
62              /// </param>
63              /// <returns>
64              /// <para>The bool</para>
65              /// <para></para>
66              /// </returns>
67              [MethodImpl(MethodImplOptions.AggressiveInlining)]
68              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);
69
70              /// <summary>
71              /// <para>
72              /// Gets the hash code using the specified pair.
73              /// </para>
74              /// <para></para>
75              /// </summary>
76              /// <param name="pair">
77              /// <para>The pair.</para>
78              /// <para></para>
79              /// </param>
80              /// <returns>
81              /// <para>The int</para>
82              /// <para></para>
83              /// </returns>
84              [MethodImpl(MethodImplOptions.AggressiveInlining)]
85              public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
                ↪  (_listComparer.GetHashCode(pair.Key),
                ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
86          }
87
88          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
89          {
90              private readonly IListComparer<TLink> _listComparer;
91
92              /// <summary>
93              /// <para>
94              /// Initializes a new <see cref="ItemComparer"/> instance.
95              /// </para>
96              /// <para></para>
97              /// </summary>
98              [MethodImpl(MethodImplOptions.AggressiveInlining)]
99              public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
100
101             /// <summary>
102             /// <para>
103             /// Compares the left.
104             /// </para>
105             /// <para></para>
106             /// </summary>
107             /// <param name="left">
108             /// <para>The left.</para>
109             /// <para></para>
110             /// </param>
111             /// <param name="right">
112             /// <para>The right.</para>
113             /// <para></para>
114             /// </param>
115             /// <returns>
116             /// <para>The intermediate result.</para>
117             /// <para></para>
118             /// </returns>
119             [MethodImpl(MethodImplOptions.AggressiveInlining)]
120             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right)
121             {
122                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
123                 if (intermediateResult == 0)
124                 {
125                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
126                 }
127                 return intermediateResult;
128             }
129         }
130
131         /// <summary>
132         /// <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>
        /// Gets this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The result list.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var links = _links;
            var count = links.Count();
            _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
            links.Each(link =>
            {
                var linkIndex = links.GetIndex(link);
                var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
                var constants = links.Constants;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
                    _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                    ↪  LinkAddress<TLink>(linkIndex));
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return constants.Continue;
            });
            var resultList = _groups.ToList();
            var comparer = Default<ItemComparer>.Instance;
            resultList.Sort(comparer);
#if DEBUG
            foreach (var item in resultList)
            {
                PrintDuplicates(item);
            }
#endif
            return resultList;
        }

        /// <summary>
        /// <para>
        /// Creates the segment using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
```

```
209             /// <param name="offset">
210             /// <para>The offset.</para>
211             /// <para></para>
212             /// </param>
213             /// <param name="length">
214             /// <para>The length.</para>
215             /// <para></para>
216             /// </param>
217             /// <returns>
218             /// <para>A segment of t link</para>
219             /// <para></para>
220             /// </returns>
221             [MethodImpl(MethodImplOptions.AggressiveInlining)]
222             protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
                ↪  length) => new Segment<TLink>(elements, offset, length);
223
224             /// <summary>
225             /// <para>
226             /// Ons the dublicate found using the specified segment.
227             /// </para>
228             /// <para></para>
229             /// </summary>
230             /// <param name="segment">
231             /// <para>The segment.</para>
232             /// <para></para>
233             /// </param>
234             [MethodImpl(MethodImplOptions.AggressiveInlining)]
235             protected override void OnDublicateFound(Segment<TLink> segment)
236             {
237                 var duplicates = CollectDuplicatesForSegment(segment);
238                 if (duplicates.Count > 1)
239                 {
240                     _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                        ↪  duplicates));
241                 }
242             }
243
244             /// <summary>
245             /// <para>
246             /// Collects the duplicates for segment using the specified segment.
247             /// </para>
248             /// <para></para>
249             /// </summary>
250             /// <param name="segment">
251             /// <para>The segment.</para>
252             /// <para></para>
253             /// </param>
254             /// <returns>
255             /// <para>The duplicates.</para>
256             /// <para></para>
257             /// </returns>
258             [MethodImpl(MethodImplOptions.AggressiveInlining)]
259             private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
260             {
261                 var duplicates = new List<TLink>();
262                 var readAsElement = new HashSet<TLink>();
263                 var restrictions = segment.ShiftRight();
264                 var constants = _links.Constants;
265                 restrictions[0] = constants.Any;
266                 _sequences.Each(sequence =>
267                 {
268                     var sequenceIndex = sequence[constants.IndexPart];
269                     duplicates.Add(sequenceIndex);
270                     readAsElement.Add(sequenceIndex);
271                     return constants.Continue;
272                 }, restrictions);
273                 if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
274                 {
275                     return new List<TLink>();
276                 }
277                 foreach (var duplicate in duplicates)
278                 {
279                     var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
280                     _visited.Set(duplicateBitIndex);
281                 }
282                 if (_sequences is Sequences sequencesExperiments)
283                 {
```

```
                var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((HashSet<ulong>)(object)readAsElement, (IList<ulong>)segment);
                foreach (var partiallyMatchedSequence in partiallyMatched)
                {
                    var sequenceIndex = _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
                    duplicates.Add(sequenceIndex);
                }
            }
            duplicates.Sort();
            return duplicates;
        }

        /// <summary>
        /// <para>
        /// Prints the duplicates using the specified duplicates item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="duplicatesItem">
        /// <para>The duplicates item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
        {
            if (!(_links is ILinks<ulong> ulongLinks))
            {
                return;
            }
            var duplicatesKey = duplicatesItem.Key;
            var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
            Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
            var duplicatesList = duplicatesItem.Value;
            for (int i = 0; i < duplicatesList.Count; i++)
            {
                var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
                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));
                Console.WriteLine(formatedSequenceStructure);
                var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex, ulongLinks);
                Console.WriteLine(sequenceString);
            }
            Console.WriteLine();
        }
    }
}
```

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

```
using System;
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.Cache
{
    /// <remarks>
    /// Can be used to operate with many CompressingConverters (to keep global frequencies data between them).
    /// TODO: Extract interface to implement frequencies storage inside Links storage
    /// </remarks>
    public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;

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

        private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
        private readonly ICounter<TLink, TLink> _frequencyCounter;
```

```
25
26          /// <summary>
27          /// <para>
28          /// Initializes a new <see cref="LinkFrequenciesCache"/> instance.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          /// <param name="links">
33          /// <para>A links.</para>
34          /// <para></para>
35          /// </param>
36          /// <param name="frequencyCounter">
37          /// <para>A frequency counter.</para>
38          /// <para></para>
39          /// </param>
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
42              : base(links)
43          {
44              _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
                ↪  DoubletComparer<TLink>.Default);
45              _frequencyCounter = frequencyCounter;
46          }
47
48          /// <summary>
49          /// <para>
50          /// Gets the frequency using the specified source.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          /// <param name="source">
55          /// <para>The source.</para>
56          /// <para></para>
57          /// </param>
58          /// <param name="target">
59          /// <para>The target.</para>
60          /// <para></para>
61          /// </param>
62          /// <returns>
63          /// <para>A link frequency of t link</para>
64          /// <para></para>
65          /// </returns>
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
68          {
69              var doublet = new Doublet<TLink>(source, target);
70              return GetFrequency(ref doublet);
71          }
72
73          /// <summary>
74          /// <para>
75          /// Gets the frequency using the specified doublet.
76          /// </para>
77          /// <para></para>
78          /// </summary>
79          /// <param name="doublet">
80          /// <para>The doublet.</para>
81          /// <para></para>
82          /// </param>
83          /// <returns>
84          /// <para>The data.</para>
85          /// <para></para>
86          /// </returns>
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
89          {
90              _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
91              return data;
92          }
93
94          /// <summary>
95          /// <para>
96          /// Increments the frequencies using the specified sequence.
97          /// </para>
98          /// <para></para>
99          /// </summary>
100         /// <param name="sequence">
101         /// <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>
    {
        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>
    {
        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>
```

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

```
90          public virtual TLink Count()
91          {
92              if (_comparer.Compare(_total, default) > 0)
93              {
94                  return _total;
95              }
96              StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
                ↪  IsElement, VisitElement);
97              return _total;
98          }
99
100         /// <summary>
101         /// <para>
102         /// Determines whether this instance is element.
103         /// </para>
104         /// <para></para>
105         /// </summary>
106         /// <param name="x">
107         /// <para>The .</para>
108         /// <para></para>
109         /// </param>
110         /// <returns>
111         /// <para>The bool</para>
112         /// <para></para>
113         /// </returns>
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
            ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
            ↪  IsPartialPoint
116
117         /// <summary>
118         /// <para>
119         /// Determines whether this instance visit element.
120         /// </para>
121         /// <para></para>
122         /// </summary>
123         /// <param name="element">
124         /// <para>The element.</para>
125         /// <para></para>
126         /// </param>
127         /// <returns>
128         /// <para>The bool</para>
129         /// <para></para>
130         /// </returns>
131         [MethodImpl(MethodImplOptions.AggressiveInlining)]
132         private bool VisitElement(TLink element)
133         {
134             if (_equalityComparer.Equals(element, _symbol))
135             {
136                 _total = Arithmetic.Increment(_total);
137             }
138             return true;
139         }
140     }
141 }
```

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the total marked sequence symbol frequency counter.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="ICounter{TLink, TLink}"/>
15     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
16     {
17         private readonly ILinks<TLink> _links;
18         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
19
20         /// <summary>
21         /// <para>
22         /// 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>
    {
        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.c

```
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>
    {
        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>
    /// Represents the total sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLink}"/>
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        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 symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLink _symbol;
        /// <summary>
        /// <para>
        /// The visits.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly HashSet<TLink> _visits;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
        /// <para></para>
        /// </summary>
        protected TLink _total;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalSequenceSymbolFrequencyOneOffCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLink>();
            _total = default;
        }

        /// <summary>
```

```
75          /// <para>
76          /// Counts this instance.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <returns>
81          /// <para>The total.</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public TLink Count()
86          {
87              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
88              {
89                  return _total;
90              }
91              CountCore(_symbol);
92              return _total;
93          }
94
95          /// <summary>
96          /// <para>
97          /// Counts the core using the specified link.
98          /// </para>
99          /// <para></para>
100         /// </summary>
101         /// <param name="link">
102         /// <para>The link.</para>
103         /// <para></para>
104         /// </param>
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         private void CountCore(TLink link)
107         {
108             var any = _links.Constants.Any;
109             if (_equalityComparer.Equals(_links.Count(any, link), default))
110             {
111                 CountSequenceSymbolFrequency(link);
112             }
113             else
114             {
115                 _links.Each(EachElementHandler, any, link);
116             }
117         }
118
119         /// <summary>
120         /// <para>
121         /// Counts the sequence symbol frequency using the specified link.
122         /// </para>
123         /// <para></para>
124         /// </summary>
125         /// <param name="link">
126         /// <para>The link.</para>
127         /// <para></para>
128         /// </param>
129         [MethodImpl(MethodImplOptions.AggressiveInlining)]
130         protected virtual void CountSequenceSymbolFrequency(TLink link)
131         {
132             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  link, _symbol);
133             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
134         }
135
136         /// <summary>
137         /// <para>
138         /// Eaches the element handler using the specified doublet.
139         /// </para>
140         /// <para></para>
141         /// </summary>
142         /// <param name="doublet">
143         /// <para>The doublet.</para>
144         /// <para></para>
145         /// </param>
146         /// <returns>
147         /// <para>The link</para>
148         /// <para></para>
149         /// </returns>
150         [MethodImpl(MethodImplOptions.AggressiveInlining)]
151         private TLink EachElementHandler(IList<TLink> doublet)
```

```
        {
            var constants = _links.Constants;
            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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        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>
    {
        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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        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)]
        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
        ↪  IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

        /// <summary>
        /// <para>
        /// Increments the frequency.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="frequency">
        /// <para>The frequency.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Increment(TLink frequency)
        {
            var links = _links;
            if (_equalityComparer.Equals(frequency, default))
            {
                return links.GetOrCreate(_unaryOne, _frequencyMarker);
            }
            var incrementedSource =
            ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
            return links.GetOrCreate(incrementedSource, _frequencyMarker);
        }
    }
}
```

1.24 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.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 unary number incrementer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IIncrementer{TLink}"/>
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly TLink _unaryOne;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnaryNumberIncrementer"/> 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 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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        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;
        }

        /// <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 link = _links.SearchOrDefault(source, target);
            var indexed = !_equalityComparer.Equals(link, default);
            if (indexed)
            {
                Increment(link);
            }
            return indexed;
```

```
106         }
107
108         /// <summary>
109         /// <para>
110         /// Increments the link.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="link">
115         /// <para>The link.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         private void Increment(TLink link)
120         {
121             var previousFrequency = _frequencyPropertyOperator.Get(link);
122             var frequency = _frequencyIncrementer.Increment(previousFrequency);
123             _frequencyPropertyOperator.Set(link, frequency);
124         }
125     }
126 }
```

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>
28         /// <para></para>
29         /// </summary>
30         /// <param name="sequence">
31         /// <para>The sequence.</para>
32         /// <para></para>
33         /// </param>
34         /// <returns>
35         /// <para>The bool</para>
36         /// <para></para>
37         /// </returns>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         bool MightContain(IList<TLink> sequence);
40     }
41 }
```

1.28 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.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     /// Represents the sequence index.
11     /// </para>
12     /// <para></para>
```

```
13      /// </summary>
14      /// <seealso cref="LinksOperatorBase{TLink}"/>
15      /// <seealso cref="ISequenceIndex{TLink}"/>
16      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
17      {
18          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
19
20          /// <summary>
21          /// <para>
22          /// Initializes a new <see cref="SequenceIndex"/> instance.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="links">
27          /// <para>A links.</para>
28          /// <para></para>
29          /// </param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public SequenceIndex(ILinks<TLink> links) : base(links) { }
32
33          /// <summary>
34          /// <para>
35          /// Determines whether this instance add.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="sequence">
40          /// <para>The sequence.</para>
41          /// <para></para>
42          /// </param>
43          /// <returns>
44          /// <para>The indexed.</para>
45          /// <para></para>
46          /// </returns>
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public virtual bool Add(IList<TLink> sequence)
49          {
50              var indexed = true;
51              var i = sequence.Count;
52              while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪  default))) { }
53              for (; i >= 1; i--)
54              {
55                  _links.GetOrCreate(sequence[i - 1], sequence[i]);
56              }
57              return indexed;
58          }
59
60          /// <summary>
61          /// <para>
62          /// Determines whether this instance might contain.
63          /// </para>
64          /// <para></para>
65          /// </summary>
66          /// <param name="sequence">
67          /// <para>The sequence.</para>
68          /// <para></para>
69          /// </param>
70          /// <returns>
71          /// <para>The indexed.</para>
72          /// <para></para>
73          /// </returns>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public virtual bool MightContain(IList<TLink> sequence)
76          {
77              var indexed = true;
78              var i = sequence.Count;
79              while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪  default))) { }
80              return indexed;
81          }
82      }
83  }
```

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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        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}"/>
    public class RationalToDecimalConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪ IConverter<TLink, decimal>
        where TLink: struct
    {
        /// <summary>
        /// <para>
        /// The raw number sequence to big integer converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly RawNumberSequenceToBigIntegerConverter<TLink>
        ↪ RawNumberSequenceToBigIntegerConverter;

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

        /// <summary>
        /// <para>
        /// Converts the rational number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rationalNumber">
        /// <para>The rational number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <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}"/>
20      /// <seealso cref="IConverter{BigInteger, TLink}"/>
21      public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
        ↪  IConverter<BigInteger, TLink>
22          where TLink : struct
23      {
24          /// <summary>
25          /// <para>
26          /// The max value.
27          /// </para>
28          /// <para></para>
29          /// </summary>
30          public static readonly TLink MaximumValue = NumericType<TLink>.MaxValue;
31          /// <summary>
32          /// <para>
33          /// The maximum value.
34          /// </para>
35          /// <para></para>
36          /// </summary>
37          public static readonly TLink BitMask = Bit.ShiftRight(MaximumValue, 1);
38          /// <summary>
39          /// <para>
40          /// The address to number converter.
41          /// </para>
42          /// <para></para>
43          /// </summary>
44          public readonly IConverter<TLink> AddressToNumberConverter;
45          /// <summary>
46          /// <para>
47          /// The list to sequence converter.
48          /// </para>
49          /// <para></para>
50          /// </summary>
51          public readonly IConverter<IList<TLink>, TLink> ListToSequenceConverter;
52          /// <summary>
53          /// <para>
54          /// The negative number marker.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          public readonly TLink NegativeNumberMarker;
59
60          /// <summary>
61          /// <para>
62          /// Initializes a new <see cref="BigIntegerToRawNumberSequenceConverter"/> instance.
63          /// </para>
64          /// <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;
        }

        private List<TLink> GetRawNumberParts(BigInteger bigInteger)
        {
            List<TLink> rawNumbers = new();
            BigInteger currentBigInt = bigInteger;
            do
            {
                var bigIntBytes = currentBigInt.ToByteArray();
                var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLink>(), BitMask);
                var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
                rawNumbers.Add(rawNumber);
                currentBigInt >>= 63;
            }
            while (currentBigInt > 0);
            return rawNumbers;
        }

        /// <summary>
        /// <para>
        /// Converts the big integer.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="bigInteger">
        /// <para>The big integer.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public TLink Convert(BigInteger bigInteger)
        {
            var sign = bigInteger.Sign;
            var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
            ↪  bigInteger);
            var numberSequence = ListToSequenceConverter.Convert(number);
            return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
            ↪  numberSequence;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;

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

```

```
namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the long raw number sequence to number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TSource}"/>
    /// <seealso cref="IConverter{TSource, TTarget}"/>
    public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
    ↪ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
    {
        private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪ UncheckedConverter<TSource, TTarget>.Default;

        private readonly IConverter<TSource> _numberToAddressConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LongRawNumberSequenceToNumberConverter"/> 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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
        ↪ numberToAddressConverter) : base(links) => _numberToAddressConverter =
        ↪ numberToAddressConverter;

        /// <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)
        {
            var constants = Links.Constants;
            var externalReferencesRange = constants.ExternalReferencesRange;
            if (externalReferencesRange.HasValue &&
            ↪ externalReferencesRange.Value.Contains(source))
            {
                return
                ↪ _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
            }
            else
            {
                var pair = Links.GetLink(source);
                var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
                ↪ (link) => externalReferencesRange.HasValue &&
                ↪ externalReferencesRange.Value.Contains(link));
                TTarget result = default;
                foreach (var element in walker.Walk(source))
                {
                    result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
                }
                return result;
            }
        }
    }
```

```
}
```

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>
    {
        private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
        private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
        private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
        private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
        ↪  NumericType<TTarget>.BitsSize + 1);
        private static readonly TSource _maximumConvertableAddress = CheckedConverter<TTarget,
        ↪  TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪  UncheckedConverter<TSource, TTarget>.Default;

        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
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly EqualityComparer<TLink> EqualityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLink, TLink> NumberToAddressConverter;
        /// <summary>
        /// <para>
        /// The left sequence walker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly LeftSequenceWalker<TLink> LeftSequenceWalker;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLink NegativeNumberMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RawNumberSequenceToBigIntegerConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <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;
101             if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
                ↪  NegativeNumberMarker))
102             {
103                 sign = -1;
104                 bigIntegerSequence = _links.GetTarget(bigInteger);
105             }
106             using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
107             if (!enumerator.MoveNext())
108             {
109                 throw new Exception("Raw number sequence cannot be empty.");
110             }
111             var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
112             BigInteger currentBigInt = new(nextPart.ToBytes());
113             while (enumerator.MoveNext())
114             {
115                 currentBigInt <<= 63;
116                 nextPart = NumberToAddressConverter.Convert(enumerator.Current);
117                 currentBigInt |= new BigInteger(nextPart.ToBytes());
118             }
119             return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
120         }
121     }
122 }
```

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

```
1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the address to unary number converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLink}"/>
```

```
18      /// <seealso cref="IConverter{TLink}"/>
19      public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink>
20      {
21          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25          private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
26
27          /// <summary>
28          /// <para>
29          /// Initializes a new <see cref="AddressToUnaryNumberConverter"/> instance.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          /// <param name="links">
34          /// <para>A links.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="powerOf2ToUnaryNumberConverter">
38          /// <para>A power of to unary number converter.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
            ↪  powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
            ↪  powerOf2ToUnaryNumberConverter;
43
44          /// <summary>
45          /// <para>
46          /// Converts the number.
47          /// </para>
48          /// <para></para>
49          /// </summary>
50          /// <param name="number">
51          /// <para>The number.</para>
52          /// <para></para>
53          /// </param>
54          /// <returns>
55          /// <para>The target.</para>
56          /// <para></para>
57          /// </returns>
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public TLink Convert(TLink number)
60          {
61              var links = _links;
62              var nullConstant = links.Constants.Null;
63              var target = nullConstant;
64              for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
                ↪  NumericType<TLink>.BitsSize; i++)
65              {
66                  if (_equalityComparer.Equals(Bit.And(number, _one), _one))
67                  {
68                      target = _equalityComparer.Equals(target, nullConstant)
69                          ? _powerOf2ToUnaryNumberConverter.Convert(i)
70                          : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
71                  }
72                  number = Bit.ShiftRight(number, 1);
73              }
74              return target;
75          }
76      }
77  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4   using Platform.Converters;
5   using System.Runtime.CompilerServices;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Numbers.Unary
10  {
11      /// <summary>
12      /// <para>
```

```
13      /// Represents the link to its frequency number conveter.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      /// <seealso cref="LinksOperatorBase{TLink}"/>
18      /// <seealso cref="IConverter{Doublet{TLink}, TLink}"/>
19      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<Doublet<TLink>, TLink>
20      {
21          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
22
23          private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
24          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
25
26          /// <summary>
27          /// <para>
28          /// Initializes a new <see cref="LinkToItsFrequencyNumberConveter"/> instance.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          /// <param name="links">
33          /// <para>A links.</para>
34          /// <para></para>
35          /// </param>
36          /// <param name="frequencyPropertyOperator">
37          /// <para>A frequency property operator.</para>
38          /// <para></para>
39          /// </param>
40          /// <param name="unaryNumberToAddressConverter">
41          /// <para>A unary number to address converter.</para>
42          /// <para></para>
43          /// </param>
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public LinkToItsFrequencyNumberConveter(
46              ILinks<TLink> links,
47              IProperty<TLink, TLink> frequencyPropertyOperator,
48              IConverter<TLink> unaryNumberToAddressConverter)
49              : base(links)
50          {
51              _frequencyPropertyOperator = frequencyPropertyOperator;
52              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
53          }
54
55          /// <summary>
56          /// <para>
57          /// Converts the doublet.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          /// <param name="doublet">
62          /// <para>The doublet.</para>
63          /// <para></para>
64          /// </param>
65          /// <exception cref="ArgumentException">
66          /// <para>Link ({doublet}) not found. </para>
67          /// <para></para>
68          /// </exception>
69          /// <returns>
70          /// <para>The link</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public TLink Convert(Doublet<TLink> doublet)
75          {
76              var links = _links;
77              var link = links.SearchOrDefault(doublet.Source, doublet.Target);
78              if (_equalityComparer.Equals(link, default))
79              {
80                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
81              }
82              var frequency = _frequencyPropertyOperator.Get(link);
83              if (_equalityComparer.Equals(frequency, default))
84              {
85                  return default;
86              }
87              var frequencyNumber = links.GetSource(frequency);
88              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
89          }
```

```
    }
}
```

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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
        ↪  UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
        ↪  UncheckedConverter<ulong, TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);

        private readonly Dictionary<TLink, TLink> _unaryToUInt64;
        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)]
```

```
149         private static TLink Double(TLink number) =>
            ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
150     }
151 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Reflection;
4  using Platform.Converters;
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.Numbers.Unary
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the unary number to address or operation converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLink}"/>
18     /// <seealso cref="IConverter{TLink}"/>
19     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
       ↪ IConverter<TLink>
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
22         private static readonly TLink _zero = default;
23         private static readonly TLink _one = Arithmetic.Increment(_zero);
24
25         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
26
27         /// <summary>
28         /// <para>
29         /// Initializes a new <see cref="UnaryNumberToAddressOrOperationConverter"/> instance.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         /// <param name="links">
34         /// <para>A links.</para>
35         /// <para></para>
36         /// </param>
37         /// <param name="powerOf2ToUnaryNumberConverter">
38         /// <para>A power of to unary number converter.</para>
39         /// <para></para>
40         /// </param>
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
           ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
           ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
43
44         /// <summary>
45         /// <para>
46         /// Converts the source number.
47         /// </para>
48         /// <para></para>
49         /// </summary>
50         /// <param name="sourceNumber">
51         /// <para>The source number.</para>
52         /// <para></para>
53         /// </param>
54         /// <returns>
55         /// <para>The target.</para>
56         /// <para></para>
57         /// </returns>
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public TLink Convert(TLink sourceNumber)
60         {
61             var links = _links;
62             var nullConstant = links.Constants.Null;
63             var source = sourceNumber;
64             var target = nullConstant;
65             if (!_equalityComparer.Equals(source, nullConstant))
66             {
67                 while (true)
68                 {
69                     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>
```

```
170         [MethodImpl(MethodImplOptions.AggressiveInlining)]
171         private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
172         {
173             if (sequence.Length == 2)
174             {
175                 var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
176                 if (link == Constants.Null)
177                 {
178                     throw new NotImplementedException("Creation cancellation is not
                        ↪ implemented.");
179                 }
180                 results.Add(link);
181                 return results;
182             }
183             var innerSequenceLength = sequence.Length - 1;
184             var innerSequence = new ulong[innerSequenceLength];
185             for (var li = 0; li < innerSequenceLength; li++)
186             {
187                 var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
188                 if (link == Constants.Null)
189                 {
190                     throw new NotImplementedException("Creation cancellation is not
                        ↪ implemented.");
191                 }
192                 for (var isi = 0; isi < li; isi++)
193                 {
194                     innerSequence[isi] = sequence[isi];
195                 }
196                 innerSequence[li] = link;
197                 for (var isi = li + 1; isi < innerSequenceLength; isi++)
198                 {
199                     innerSequence[isi] = sequence[isi + 1];
200                 }
201                 CreateAllVariants1Core(innerSequence, results);
202             }
203             return results;
204         }
205 
206         #endregion
207 
208         /// <summary>
209         /// <para>
210         /// Eaches the 1 using the specified sequence.
211         /// </para>
212         /// <para></para>
213         /// </summary>
214         /// <param name="sequence">
215         /// <para>The sequence.</para>
216         /// <para></para>
217         /// </param>
218         /// <returns>
219         /// <para>The visited links.</para>
220         /// <para></para>
221         /// </returns>
222         [MethodImpl(MethodImplOptions.AggressiveInlining)]
223         public HashSet<ulong> Each1(params ulong[] sequence)
224         {
225             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
226             Each1(link =>
227             {
228                 if (!visitedLinks.Contains(link))
229                 {
230                     visitedLinks.Add(link); // изучить почему случаются повторы
231                 }
232                 return true;
233             }, sequence);
234             return visitedLinks;
235         }
236 
237         /// <summary>
238         /// <para>
239         /// Eaches the 1 using the specified handler.
240         /// </para>
241         /// <para></para>
242         /// </summary>
243         /// <param name="handler">
244         /// <para>The handler.</para>
245         /// <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.
```

```
556         /// </para>
557         /// <para></para>
558         /// </summary>
559         /// <param name="handler">
560         /// <para>The handler.</para>
561         /// <para></para>
562         /// </param>
563         /// <param name="left">
564         /// <para>The left.</para>
565         /// <para></para>
566         /// </param>
567         /// <param name="right">
568         /// <para>The right.</para>
569         /// <para></para>
570         /// </param>
571         [MethodImpl(MethodImplOptions.AggressiveInlining)]
572         private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
573         {
574             Links.Unsync.Each(Constants.Any, right, leftStep =>
575             {
576                 TryStepLeftUp(handler, left, leftStep);
577                 return true;
578             });
579         }
580 
581         /// <summary>
582         /// <para>
583         /// Tries the step left up using the specified handler.
584         /// </para>
585         /// <para></para>
586         /// </summary>
587         /// <param name="handler">
588         /// <para>The handler.</para>
589         /// <para></para>
590         /// </param>
591         /// <param name="left">
592         /// <para>The left.</para>
593         /// <para></para>
594         /// </param>
595         /// <param name="stepFrom">
596         /// <para>The step from.</para>
597         /// <para></para>
598         /// </param>
599         [MethodImpl(MethodImplOptions.AggressiveInlining)]
600         private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
601         {
602             var upStep = stepFrom;
603             var firstTarget = Links.Unsync.GetSource(upStep);
604             while (firstTarget != left && firstTarget != upStep)
605             {
606                 upStep = firstTarget;
607                 firstTarget = Links.Unsync.GetTarget(upStep);
608             }
609             if (firstTarget == left)
610             {
611                 handler(new LinkAddress<LinkIndex>(stepFrom));
612             }
613         }
614 
615         /// <summary>
616         /// <para>
617         /// Determines whether this instance starts with.
618         /// </para>
619         /// <para></para>
620         /// </summary>
621         /// <param name="sequence">
622         /// <para>The sequence.</para>
623         /// <para></para>
624         /// </param>
625         /// <param name="link">
626         /// <para>The link.</para>
627         /// <para></para>
628         /// </param>
629         /// <returns>
630         /// <para>The bool</para>
631         /// <para></para>
632         /// </returns>
633         [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">
        /// <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 FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
        ↪  elementToString, bool insertComma, params LinkIndex[] knownElements) =>
        ↪  Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
        ↪  elementToString, insertComma, knownElements));

        /// <summary>
        /// <para>
        /// Formats the 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 FormatSequence(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.IsPartialPoint(x), element => //
                    ↪  entered.AddAndReturnVoid, x => { }, entered.DoNotContains
                    {
                        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>
        /// 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);

    //            var filteredResults = new HashSet<ulong>();
    //            var matcher = new Matcher(this, sequence, filteredResults, null);
    //            matcher.AddAllPartialMatchedToResults(firstResults);
    //            return filteredResults;
    //        }
```

```
        //        return new HashSet<ulong>();
        //    });
        //}

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences 3 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> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
                    var firstResults = new HashSet<ulong>();
                    var lastResults = new HashSet<ulong>();
                    var first = sequence.First(x => x != Constants.Any);
                    var last = sequence.Last(x => x != Constants.Any);
                    AllUsagesCore(first, firstResults);
                    AllUsagesCore(last, lastResults);
                    firstResults.IntersectWith(lastResults);
                    //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(firstResults);
                    return filteredResults;
                }
                return new HashSet<ulong>();
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences 4 using the specified read as elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="readAsElements">
        /// <para>The read as elements.</para>
        /// <para></para>
        /// </param>
        /// <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> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
        ↪  IList<ulong> sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Count > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    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)
                    //{
```

```
1386                    //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
1387                    //    //if (doublet != Doublets.Links.Null)
1388                    //    //    results.Add(doublet);
1389                    //    return results;
1390                    //}
1391                    //var lastElement = sequence[sequence.Length - 1];
1392                    //Func<ulong, bool> handler = x =>
1393                    //{
1394                    //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
                        ↪  results.Add(x);
1395                    //    return true;
1396                    //};
1397                    //if (sequence.Length >= 2)
1398                    //    StepRight(handler, sequence[0], sequence[1]);
1399                    //var last = sequence.Length - 2;
1400                    //for (var i = 1; i < last; i++)
1401                    //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
1402                    //if (sequence.Length >= 3)
1403                    //    StepLeft(handler, sequence[sequence.Length - 2],
                        ↪  sequence[sequence.Length - 1]);
1404                    //////if (sequence.Length == 1)
1405                    //////{
1406                    //////    throw new NotImplementedException(); // all sequences, containing
                        ↪  this element?
1407                    //////}
1408                    //////if (sequence.Length == 2)
1409                    //////{
1410                    //////    var results = new List<ulong>();
1411                    //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
1412                    //////    return results;
1413                    //////}
1414                    //////var matches = new List<List<ulong>>();
1415                    //////var last = sequence.Length - 1;
1416                    //////for (var i = 0; i < last; i++)
1417                    //////{
1418                    //////    var results = new List<ulong>();
1419                    //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
1420                    //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
1421                    //////    if (results.Count > 0)
1422                    //////        matches.Add(results);
1423                    //////    else
1424                    //////        return results;
1425                    //////    if (matches.Count == 2)
1426                    //////    {
1427                    //////        var merged = new List<ulong>();
1428                    //////        for (var j = 0; j < matches[0].Count; j++)
1429                    //////            for (var k = 0; k < matches[1].Count; k++)
1430                    //////                CloseInnerConnections(merged.Add, matches[0][j],
                        ↪  matches[1][k]);
1431                    //////        if (merged.Count > 0)
1432                    //////            matches = new List<List<ulong>> { merged };
1433                    //////        else
1434                    //////            return new List<ulong>();
1435                    //////    }
1436                    //////}
1437                    //////if (matches.Count > 0)
1438                    //////{
1439                    //////    var usages = new HashSet<ulong>();
1440                    //////    for (int i = 0; i < sequence.Length; i++)
1441                    //////    {
1442                    //////        AllUsagesCore(sequence[i], usages);
1443                    //////    }
1444                    //////    //for (int i = 0; i < matches[0].Count; i++)
1445                    //////    //    AllUsagesCore(matches[0][i], usages);
1446                    //////    //usages.UnionWith(matches[0]);
1447                    //////    return usages.ToList();
1448                    //////}
1449                    var firstLinkUsages = new HashSet<ulong>();
1450                    AllUsagesCore(sequence[0], firstLinkUsages);
1451                    firstLinkUsages.Add(sequence[0]);
1452                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                        ↪  sequence[0] }; // or all sequences, containing this element?
1453                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                        ↪  1).ToList();
1454                    var results = new HashSet<ulong>();
1455                    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();
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance all usages core 1.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="usages">
        /// <para>The usages.</para>
        /// <para></para>
        /// </param>
        /// <param name="outerHandler">
        /// <para>The outer handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
        ↪  LinkIndex> outerHandler)
        {
            bool handler(ulong doublet)
            {
                if (usages.Add(doublet))
                {
                    if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
                    {
                        return false;
                    }
                    if (!AllUsagesCore1(doublet, usages, outerHandler))
                    {
                        return false;
                    }
                }
                return true;
            }
            return Links.Unsync.Each(link, Constants.Any, handler)
                && Links.Unsync.Each(Constants.Any, link, handler);
        }

        /// <summary>
        /// <para>
        /// Calculates the all usages using the specified totals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="totals">
        /// <para>The totals.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CalculateAllUsages(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator(Links, totals);
            calculator.Calculate();
        }

        /// <summary>
        /// <para>
        /// Calculates the all usages 2 using the specified totals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="totals">
        /// <para>The totals.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CalculateAllUsages2(ulong[] totals)
        {
            var calculator = new AllUsagesCalculator2(Links, totals);
            calculator.Calculate();
        }

        private class AllUsagesCalculator
        {
```

```
1685            private readonly SynchronizedLinks<ulong> _links;
1686            private readonly ulong[] _totals;
1687
1688            /// <summary>
1689            /// <para>
1690            /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
1691            /// </para>
1692            /// <para></para>
1693            /// </summary>
1694            /// <param name="links">
1695            /// <para>A links.</para>
1696            /// <para></para>
1697            /// </param>
1698            /// <param name="totals">
1699            /// <para>A totals.</para>
1700            /// <para></para>
1701            /// </param>
1702            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1703            public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1704            {
1705                _links = links;
1706                _totals = totals;
1707            }
1708
1709            /// <summary>
1710            /// <para>
1711            /// Calculates this instance.
1712            /// </para>
1713            /// <para></para>
1714            /// </summary>
1715            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1716            public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
                ↪  CalculateCore);
1717
1718            /// <summary>
1719            /// <para>
1720            /// Determines whether this instance calculate core.
1721            /// </para>
1722            /// <para></para>
1723            /// </summary>
1724            /// <param name="link">
1725            /// <para>The link.</para>
1726            /// <para></para>
1727            /// </param>
1728            /// <returns>
1729            /// <para>The bool</para>
1730            /// <para></para>
1731            /// </returns>
1732            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1733            private bool CalculateCore(ulong link)
1734            {
1735                if (_totals[link] == 0)
1736                {
1737                    var total = 1UL;
1738                    _totals[link] = total;
1739                    var visitedChildren = new HashSet<ulong>();
1740                    bool linkCalculator(ulong child)
1741                    {
1742                        if (link != child && visitedChildren.Add(child))
1743                        {
1744                            total += _totals[child] == 0 ? 1 : _totals[child];
1745                        }
1746                        return true;
1747                    }
1748                    _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1749                    _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1750                    _totals[link] = total;
1751                }
1752                return true;
1753            }
1754        }
1755
1756        private class AllUsagesCalculator2
1757        {
1758            private readonly SynchronizedLinks<ulong> _links;
1759            private readonly ulong[] _totals;
1760
1761            /// <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">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(ulong link)
        {
            //_linksInSequence.Contains(link) ||
            return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
            ↪  link;
        }

        /// <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)
        {
            // TODO: Проработать защиту от зацикливания
            // Основано на SequenceWalker.WalkLeft
            Func<ulong, ulong> getSource = _links.Unsync.GetSource;
            Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
            Func<ulong, bool> isElement = IsElement;
            void visitLeaf(ulong parent)
            {
                if (link != parent)
                {
```

```
                    _totals[parent]++;
                }
            }
            void visitNode(ulong parent)
            {
                if (link != parent)
                {
                    _totals[parent]++;
                }
            }
            var stack = new Stack();
            var element = link;
            if (isElement(element))
            {
                visitLeaf(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        // Обработка элемента
                        if (isElement(target))
                        {
                            visitLeaf(target);
                        }
                        if (isElement(source))
                        {
                            visitLeaf(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        visitNode(element);
                        element = getTarget(element);
                    }
                }
            }
            _totals[link]++;
            return true;
        }
    }

    private class AllUsagesCollector
    {
        private readonly ILinks<ulong> _links;
        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;
            }
        }

        private class AllUsagesCollector1
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;
            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;
            }
        }
```

```
1996
1997        private class AllUsagesCollector2
1998        {
1999            private readonly ILinks<ulong> _links;
2000            private readonly BitString _usages;
2001
2002            /// <summary>
2003            /// <para>
2004            /// Initializes a new <see cref="AllUsagesCollector2"/> instance.
2005            /// </para>
2006            /// <para></para>
2007            /// </summary>
2008            /// <param name="links">
2009            /// <para>A links.</para>
2010            /// <para></para>
2011            /// </param>
2012            /// <param name="usages">
2013            /// <para>A usages.</para>
2014            /// <para></para>
2015            /// </param>
2016            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2017            public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
2018            {
2019                _links = links;
2020                _usages = usages;
2021            }
2022
2023            /// <summary>
2024            /// <para>
2025            /// Determines whether this instance collect.
2026            /// </para>
2027            /// <para></para>
2028            /// </summary>
2029            /// <param name="link">
2030            /// <para>The link.</para>
2031            /// <para></para>
2032            /// </param>
2033            /// <returns>
2034            /// <para>The bool</para>
2035            /// <para></para>
2036            /// </returns>
2037            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2038            public bool Collect(ulong link)
2039            {
2040                if (_usages.Add((long)link))
2041                {
2042                    _links.Each(link, _links.Constants.Any, Collect);
2043                    _links.Each(_links.Constants.Any, link, Collect);
2044                }
2045                return true;
2046            }
2047        }
2048
2049        private class AllUsagesIntersectingCollector
2050        {
2051            private readonly SynchronizedLinks<ulong> _links;
2052            private readonly HashSet<ulong> _intersectWith;
2053            private readonly HashSet<ulong> _usages;
2054            private readonly HashSet<ulong> _enter;
2055
2056            /// <summary>
2057            /// <para>
2058            /// Initializes a new <see cref="AllUsagesIntersectingCollector"/> instance.
2059            /// </para>
2060            /// <para></para>
2061            /// </summary>
2062            /// <param name="links">
2063            /// <para>A links.</para>
2064            /// <para></para>
2065            /// </param>
2066            /// <param name="intersectWith">
2067            /// <para>A intersect with.</para>
2068            /// <para></para>
2069            /// </param>
2070            /// <param name="usages">
2071            /// <para>A usages.</para>
2072            /// <para></para>
2073            /// </param>
2074            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
2075            public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
                ↪  intersectWith, HashSet<ulong> usages)
2076            {
2077                _links = links;
2078                _intersectWith = intersectWith;
2079                _usages = usages;
2080                _enter = new HashSet<ulong>(); // защита от зацикливания
2081            }
2082
2083            /// <summary>
2084            /// <para>
2085            /// Determines whether this instance collect.
2086            /// </para>
2087            /// <para></para>
2088            /// </summary>
2089            /// <param name="link">
2090            /// <para>The link.</para>
2091            /// <para></para>
2092            /// </param>
2093            /// <returns>
2094            /// <para>The bool</para>
2095            /// <para></para>
2096            /// </returns>
2097            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2098            public bool Collect(ulong link)
2099            {
2100                if (_enter.Add(link))
2101                {
2102                    if (_intersectWith.Contains(link))
2103                    {
2104                        _usages.Add(link);
2105                    }
2106                    _links.Unsync.Each(link, _links.Constants.Any, Collect);
2107                    _links.Unsync.Each(_links.Constants.Any, link, Collect);
2108                }
2109                return true;
2110            }
2111        }
2112
2113        /// <summary>
2114        /// <para>
2115        /// Closes the inner connections using the specified handler.
2116        /// </para>
2117        /// <para></para>
2118        /// </summary>
2119        /// <param name="handler">
2120        /// <para>The handler.</para>
2121        /// <para></para>
2122        /// </param>
2123        /// <param name="left">
2124        /// <para>The left.</para>
2125        /// <para></para>
2126        /// </param>
2127        /// <param name="right">
2128        /// <para>The right.</para>
2129        /// <para></para>
2130        /// </param>
2131        [MethodImpl(MethodImplOptions.AggressiveInlining)]
2132        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
                ↪  right)
2133        {
2134            TryStepLeftUp(handler, left, right);
2135            TryStepRightUp(handler, right, left);
2136        }
2137
2138        /// <summary>
2139        /// <para>
2140        /// Alls the close connections using the specified handler.
2141        /// </para>
2142        /// <para></para>
2143        /// </summary>
2144        /// <param name="handler">
2145        /// <para>The handler.</para>
2146        /// <para></para>
2147        /// </param>
2148        /// <param name="left">
2149        /// <para>The left.</para>
2150        /// <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);
            PartialStepLeft(handler, left, right);
        }

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences core using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="previousMatchings">
        /// <para>The previous matchings.</para>
        /// <para></para>
        /// </param>
        /// <param name="startAt">
        /// <para>The start at.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪  HashSet<ulong> previousMatchings, long startAt)
        {
            if (startAt >= sequence.Length) // ?
            {
                return previousMatchings;
            }
            var secondLinkUsages = new HashSet<ulong>();
            AllUsagesCore(sequence[startAt], secondLinkUsages);
            secondLinkUsages.Add(sequence[startAt]);
            var matchings = new HashSet<ulong>();
            var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
            //for (var i = 0; i < previousMatchings.Count; i++)
            foreach (var secondLinkUsage in secondLinkUsages)
            {
                foreach (var previousMatching in previousMatchings)
                {
                    //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
                    ↪  secondLinkUsage);
                    StepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪  secondLinkUsage);
                    TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
                    ↪  previousMatching);
                    //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
                    ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к
                    ↪  желаемым результам.
```

```
                    PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
                    ↪  secondLinkUsage);
                }
            }
            if (matchings.Count == 0)
            {
                return matchings;
            }
            return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
        }

        /// <summary>
        /// <para>
        /// Ensures the each link is any or zero or many or exists using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </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>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
                        collector.Collect(linksToConnect[i]);
                        //AllUsagesCore(linksToConnect[i], next);
                        //results.IntersectWith(next);
                        results = next;
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all connections 3 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 list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                ↪  BitArray((int)_links.Total + 1);
```

```
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                        ↪  BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }

        /// <summary>
        /// <para>
        /// Simplifies the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <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>
        {
            private readonly Sequences _sequences;
            private readonly ulong[] _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            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;
            }

            private readonly List<PatternBlock> _pattern;
            private int _patternPosition;
            private long _sequencePosition;

            #endregion

            /// <summary>
```

```
        /// <para>
        /// Initializes a new <see cref="PatternMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequences">
        /// <para>A sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="patternSequence">
        /// <para>A pattern sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="results">
        /// <para>A results.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
        ↪  HashSet<LinkIndex> results)
            : base(sequences.Links.Unsync, new DefaultStack<ulong>())
        {
            _sequences = sequences;
            _patternSequence = patternSequence;
            _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
            ↪  _sequences.Constants.Any && x != ZeroOrMany));
            _results = results;
            _pattern = CreateDetailedPattern();
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is element.
        /// </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)]
        protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
        ↪  base.IsElement(link);

        /// <summary>
        /// <para>
        /// Determines whether this instance pattern match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceToMatch">
        /// <para>The sequence to match.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <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++;
    //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
    //            maximumGap = long.MaxValue;
    //        else
    //            break;
    //    }

    //    if (maximumGap < mininumGap)
    //        maximumGap = mininumGap;
    //}

    /// <summary>
    /// <para>
    /// Determines whether this instance pattern match core.
    /// </para>
    /// <para></para>
    /// </summary>
```

```
3219            /// <param name="element">
3220            /// <para>The element.</para>
3221            /// <para></para>
3222            /// </param>
3223            /// <returns>
3224            /// <para>The bool</para>
3225            /// <para></para>
3226            /// </returns>
3227            [MethodImpl(MethodImplOptions.AggressiveInlining)]
3228            private bool PatternMatchCore(LinkIndex element)
3229            {
3230                if (_patternPosition >= _pattern.Count)
3231                {
3232                    _patternPosition = -2;
3233                    return false;
3234                }
3235                var currentPatternBlock = _pattern[_patternPosition];
3236                if (currentPatternBlock.Type == PatternBlockType.Gap)
3237                {
3238                    //var currentMatchingBlockLength = (_sequencePosition -
                        ↪ _lastMatchedBlockPosition);
3239                    if (_sequencePosition < currentPatternBlock.Start)
3240                    {
3241                        _sequencePosition++;
3242                        return true; // Двигаемся дальше
3243                    }
3244                    // Это последний блок
3245                    if (_pattern.Count == _patternPosition + 1)
3246                    {
3247                        _patternPosition++;
3248                        _sequencePosition = 0;
3249                        return false; // Полное соответствие
3250                    }
3251                    else
3252                    {
3253                        if (_sequencePosition > currentPatternBlock.Stop)
3254                        {
3255                            return false; // Соответствие невозможно
3256                        }
3257                        var nextPatternBlock = _pattern[_patternPosition + 1];
3258                        if (_patternSequence[nextPatternBlock.Start] == element)
3259                        {
3260                            if (nextPatternBlock.Start < nextPatternBlock.Stop)
3261                            {
3262                                _patternPosition++;
3263                                _sequencePosition = 1;
3264                            }
3265                            else
3266                            {
3267                                _patternPosition += 2;
3268                                _sequencePosition = 0;
3269                            }
3270                        }
3271                    }
3272                }
3273                else // currentPatternBlock.Type == PatternBlockType.Elements
3274                {
3275                    var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
3276                    if (_patternSequence[patternElementPosition] != element)
3277                    {
3278                        return false; // Соответствие невозможно
3279                    }
3280                    if (patternElementPosition == currentPatternBlock.Stop)
3281                    {
3282                        _patternPosition++;
3283                        _sequencePosition = 0;
3284                    }
3285                    else
3286                    {
3287                        _sequencePosition++;
3288                    }
3289                }
3290                return true;
3291                //if (_patternSequence[_patternPosition] != element)
3292                //    return false;
3293                //else
3294                //{
3295                //    _sequencePosition++;
3296                //    _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.
    ///
```

```
22      /// TODO:
23      ///
24      /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25      /// через естественную группировку по unicode типам, все whitespace вместе, все символы
        ↪  вместе, все числа вместе и т.п.
26      /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
        ↪  графа)
27      ///
28      /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
        ↪  ограничитель на то, что является последовательностью, а что нет,
29      /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
        ↪  порядке.
30      ///
31      /// Рост последовательности слева и справа.
32      /// Поиск со звёздочкой.
33      /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
34      /// так же проблема может быть решена при реализации дистанционных триггеров.
35      /// Нужны ли уникальные указатели вообще?
36      /// Что если обращение к информации будет происходить через содержимое всегда?
37      ///
38      /// Писать тесты.
39      ///
40      ///
41      /// Можно убрать зависимость от конкретной реализации Links,
42      /// на зависимость от абстрактного элемента, который может быть представлен несколькими
        ↪  способами.
43      ///
44      /// Можно ли как-то сделать один общий интерфейс
45      ///
46      ///
47      /// Блокчейн и/или гит для распределённой записи транзакций.
48      ///
49      /// </remarks>
50      public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
        ↪  (после завершения реализации Sequences)
51      {
52          /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
            ↪  связей.</summary>
53          public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
54
55          /// <summary>
56          /// <para>
57          /// Gets the options value.
58          /// </para>
59          /// <para></para>
60          /// </summary>
61          public SequencesOptions<LinkIndex> Options { get; }
62          /// <summary>
63          /// <para>
64          /// Gets the links value.
65          /// </para>
66          /// <para></para>
67          /// </summary>
68          public SynchronizedLinks<LinkIndex> Links { get; }
69          private readonly ISynchronization _sync;
70
71          /// <summary>
72          /// <para>
73          /// Gets the constants value.
74          /// </para>
75          /// <para></para>
76          /// </summary>
77          public LinksConstants<LinkIndex> Constants { get; }
78
79          /// <summary>
80          /// <para>
81          /// Initializes a new <see cref="Sequences"/> instance.
82          /// </para>
83          /// <para></para>
84          /// </summary>
85          /// <param name="links">
86          /// <para>A links.</para>
87          /// <para></para>
88          /// </param>
89          /// <param name="options">
90          /// <para>A options.</para>
91          /// <para></para>
92          /// </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>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The results.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<LinkIndex> Each(IList<LinkIndex> sequence)
        {
            var results = new List<LinkIndex>();
            var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
            Each(filler.AddFirstAndReturnConstant, sequence);
            return results;
        }

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

```
403             /// <para></para>
404             /// </summary>
405             /// <param name="handler">
406             /// <para>The handler.</para>
407             /// <para></para>
408             /// </param>
409             /// <param name="restrictions">
410             /// <para>The restrictions.</para>
411             /// <para></para>
412             /// </param>
413             /// <exception cref="NotImplementedException">
414             /// <para></para>
415             /// <para></para>
416             /// </exception>
417             /// <returns>
418             /// <para>The link index</para>
419             /// <para></para>
420             /// </returns>
421             [MethodImpl(MethodImplOptions.AggressiveInlining)]
422             public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
                ↪ restrictions)
423             {
424                 return _sync.ExecuteReadOperation(() =>
425                 {
426                     if (restrictions.IsNullOrEmpty())
427                     {
428                         return Constants.Continue;
429                     }
430                     Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
431                     if (restrictions.Count == 1)
432                     {
433                         var link = restrictions[0];
434                         var any = Constants.Any;
435                         if (link == any)
436                         {
437                             if (Options.UseSequenceMarker)
438                             {
439                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                    ↪ Options.SequenceMarkerLink, any));
440                             }
441                             else
442                             {
443                                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                    ↪ any));
444                             }
445                         }
446                         if (Options.UseSequenceMarker)
447                         {
448                             var sequenceLinkValues = Links.Unsync.GetLink(link);
449                             if (sequenceLinkValues[Constants.SourcePart] ==
                                ↪ Options.SequenceMarkerLink)
450                             {
451                                 link = sequenceLinkValues[Constants.TargetPart];
452                             }
453                         }
454                         var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
455                         sequence[0] = link;
456                         return handler(sequence);
457                     }
458                     else if (restrictions.Count == 2)
459                     {
460                         throw new NotImplementedException();
461                     }
462                     else if (restrictions.Count == 3)
463                     {
464                         return Links.Unsync.Each(handler, restrictions);
465                     }
466                     else
467                     {
468                         var sequence = restrictions.SkipFirst();
469                         if (Options.UseIndex && !Options.Index.MightContain(sequence))
470                         {
471                             return Constants.Break;
472                         }
473                         return EachCore(handler, sequence);
474                     }
475                 });
476             }
```

```
477
478         /// <summary>
479         /// <para>
480         /// Eaches the core using the specified handler.
481         /// </para>
482         /// <para></para>
483         /// </summary>
484         /// <param name="handler">
485         /// <para>The handler.</para>
486         /// <para></para>
487         /// </param>
488         /// <param name="values">
489         /// <para>The values.</para>
490         /// <para></para>
491         /// </param>
492         /// <returns>
493         /// <para>The link index</para>
494         /// <para></para>
495         /// </returns>
496         [MethodImpl(MethodImplOptions.AggressiveInlining)]
497         private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ values)
498         {
499             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
500             // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
                ↪ Id.
501             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
                ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
                ↪ matcher.HandleFullMatched;
502             //if (sequence.Length >= 2)
503             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
504             {
505                 return Constants.Break;
506             }
507             var last = values.Count - 2;
508             for (var i = 1; i < last; i++)
509             {
510                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                    ↪ Constants.Continue)
511                 {
512                     return Constants.Break;
513                 }
514             }
515             if (values.Count >= 3)
516             {
517                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                    ↪ != Constants.Continue)
518                 {
519                     return Constants.Break;
520                 }
521             }
522             return Constants.Continue;
523         }
524
525         /// <summary>
526         /// <para>
527         /// Partials the step right using the specified handler.
528         /// </para>
529         /// <para></para>
530         /// </summary>
531         /// <param name="handler">
532         /// <para>The handler.</para>
533         /// <para></para>
534         /// </param>
535         /// <param name="left">
536         /// <para>The left.</para>
537         /// <para></para>
538         /// </param>
539         /// <param name="right">
540         /// <para>The right.</para>
541         /// <para></para>
542         /// </param>
543         /// <returns>
544         /// <para>The link index</para>
545         /// <para></para>
546         /// </returns>
547         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
548         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ left, LinkIndex right)
549         {
550             return Links.Unsync.Each(doublet =>
551             {
552                 var doubletIndex = doublet[Constants.IndexPart];
553                 if (StepRight(handler, doubletIndex, right) != Constants.Continue)
554                 {
555                     return Constants.Break;
556                 }
557                 if (left != doubletIndex)
558                 {
559                     return PartialStepRight(handler, doubletIndex, right);
560                 }
561                 return Constants.Continue;
562             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
563         }
564 
565         /// <summary>
566         /// <para>
567         /// Steps the right using the specified handler.
568         /// </para>
569         /// <para></para>
570         /// </summary>
571         /// <param name="handler">
572         /// <para>The handler.</para>
573         /// <para></para>
574         /// </param>
575         /// <param name="left">
576         /// <para>The left.</para>
577         /// <para></para>
578         /// </param>
579         /// <param name="right">
580         /// <para>The right.</para>
581         /// <para></para>
582         /// </param>
583         /// <returns>
584         /// <para>The link index</para>
585         /// <para></para>
586         /// </returns>
587         [MethodImpl(MethodImplOptions.AggressiveInlining)]
588         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));
589 
590         /// <summary>
591         /// <para>
592         /// Tries the step right up using the specified handler.
593         /// </para>
594         /// <para></para>
595         /// </summary>
596         /// <param name="handler">
597         /// <para>The handler.</para>
598         /// <para></para>
599         /// </param>
600         /// <param name="right">
601         /// <para>The right.</para>
602         /// <para></para>
603         /// </param>
604         /// <param name="stepFrom">
605         /// <para>The step from.</para>
606         /// <para></para>
607         /// </param>
608         /// <returns>
609         /// <para>The link index</para>
610         /// <para></para>
611         /// </returns>
612         [MethodImpl(MethodImplOptions.AggressiveInlining)]
613         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
            ↪ right, LinkIndex stepFrom)
614         {
615             var upStep = stepFrom;
616             var firstSource = Links.Unsync.GetTarget(upStep);
617             while (firstSource != right && firstSource != upStep)
618             {
619                 upStep = firstSource;
620                 firstSource = Links.Unsync.GetSource(upStep);
```

```
            }
            if (firstSource == right)
            {
                return handler(new LinkAddress<LinkIndex>(stepFrom));
            }
            return Constants.Continue;
        }

        /// <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>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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));

        /// <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>
        /// <returns>
        /// <para>The link index</para>
        /// <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

```

```
848         #region Delete
849
850         /// <summary>
851         /// <para>
852         /// Deletes the restrictions.
853         /// </para>
854         /// <para></para>
855         /// </summary>
856         /// <param name="restrictions">
857         /// <para>The restrictions.</para>
858         /// <para></para>
859         /// </param>
860         [MethodImpl(MethodImplOptions.AggressiveInlining)]
861         public void Delete(IList<LinkIndex> restrictions)
862         {
863             _sync.ExecuteWriteOperation(() =>
864             {
865                 var sequence = restrictions.SkipFirst();
866                 // TODO: Check all options only ones before loop execution
867                 foreach (var linkToDelete in Each(sequence))
868                 {
869                     DeleteOneCore(linkToDelete);
870                 }
871             });
872         }
873
874         /// <summary>
875         /// <para>
876         /// Deletes the one core using the specified link.
877         /// </para>
878         /// <para></para>
879         /// </summary>
880         /// <param name="link">
881         /// <para>The link.</para>
882         /// <para></para>
883         /// </param>
884         [MethodImpl(MethodImplOptions.AggressiveInlining)]
885         private void DeleteOneCore(LinkIndex link)
886         {
887             if (Options.UseGarbageCollection)
888             {
889                 var sequenceElements = GetSequenceElements(link);
890                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
891                 var sequenceLink = GetSequenceByElements(sequenceElements);
892                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
893                 {
894                     if (sequenceLink != Constants.Null)
895                     {
896                         Links.Unsync.Delete(sequenceLink);
897                     }
898                     Links.Unsync.Delete(link);
899                 }
900                 ClearGarbage(sequenceElementsContents.Source);
901                 ClearGarbage(sequenceElementsContents.Target);
902             }
903             else
904             {
905                 if (Options.UseSequenceMarker)
906                 {
907                     var sequenceElements = GetSequenceElements(link);
908                     var sequenceLink = GetSequenceByElements(sequenceElements);
909                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
910                     {
911                         if (sequenceLink != Constants.Null)
912                         {
913                             Links.Unsync.Delete(sequenceLink);
914                         }
915                         Links.Unsync.Delete(link);
916                     }
917                 }
918                 else
919                 {
920                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
921                     {
922                         Links.Unsync.Delete(link);
923                     }
924                 }
925             }
```

```
926         }
927
928         #endregion
929
930         #region Compactification
931
932         /// <summary>
933         /// <para>
934         /// Compacts the all.
935         /// </para>
936         /// <para></para>
937         /// </summary>
938         [MethodImpl(MethodImplOptions.AggressiveInlining)]
939         public void CompactAll()
940         {
941             _sync.ExecuteWriteOperation(() =>
942             {
943                 var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
944                 for (int i = 0; i < sequences.Count; i++)
945                 {
946                     var sequence = this.ToList(sequences[i]);
947                     Compact(sequence.ShiftRight());
948                 }
949             });
950         }
951
952         /// <remarks>
953         /// bestVariant можно выбирать по максимальному числу использований,
954         /// но балансированный позволяет гарантировать уникальность (если есть возможность,
955         /// гарантировать его использование в других местах).
956         ///
957         /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
958         /// </remarks>
959         [MethodImpl(MethodImplOptions.AggressiveInlining)]
960         public LinkIndex Compact(IList<LinkIndex> sequence)
961         {
962             return _sync.ExecuteWriteOperation(() =>
963             {
964                 if (sequence.IsNullOrEmpty())
965                 {
966                     return Constants.Null;
967                 }
968                 Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
969                 return CompactCore(sequence);
970             });
971         }
972
973         /// <summary>
974         /// <para>
975         /// Compacts the core using the specified sequence.
976         /// </para>
977         /// <para></para>
978         /// </summary>
979         /// <param name="sequence">
980         /// <para>The sequence.</para>
981         /// <para></para>
982         /// </param>
983         /// <returns>
984         /// <para>The link index</para>
985         /// <para></para>
986         /// </returns>
987         [MethodImpl(MethodImplOptions.AggressiveInlining)]
988         private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
          ↪  sequence);
989
990         #endregion
991
992         #region Garbage Collection
993
994         /// <remarks>
995         /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
          ↪  определить извне или в унаследованном классе
996         /// </remarks>
997         [MethodImpl(MethodImplOptions.AggressiveInlining)]
998         private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
          ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
999
1000        /// <summary>
1001        /// <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;
            });
        }

        /// <summary>
        /// <para>
        /// Represents the matcher.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="RightSequenceWalker{LinkIndex}"/>
        public class Matcher : RightSequenceWalker<LinkIndex>
        {
            private readonly Sequences _sequences;
            private readonly IList<LinkIndex> _patternSequence;
            private readonly HashSet<LinkIndex> _linksInSequence;
            private readonly HashSet<LinkIndex> _results;
            private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
            private readonly HashSet<LinkIndex> _readAsElements;
            private int _filterPosition;

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

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequences">
        /// <para>A sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="patternSequence">
        /// <para>A pattern sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="results">
        /// <para>A results.</para>
        /// <para></para>
        /// </param>
        /// <param name="stopableHandler">
        /// <para>A stopable handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="readAsElements">
        /// <para>A read as elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
        ↪  HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
        ↪  HashSet<LinkIndex> readAsElements = null)
            : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
        {
            _sequences = sequences;
            _patternSequence = patternSequence;
            _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
            ↪  _links.Constants.Any && x != ZeroOrMany));
            _results = results;
            _stopableHandler = stopableHandler;
            _readAsElements = readAsElements;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is element.
        /// </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)]
        protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
        ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
        ↪  _linksInSequence.Contains(link);

        /// <summary>
        /// <para>
        /// Determines whether this instance full match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceToMatch">
        /// <para>The sequence to match.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool FullMatch(LinkIndex sequenceToMatch)
        {
            _filterPosition = 0;
            foreach (var part in Walk(sequenceToMatch))
            {
                if (!FullMatchCore(part))
```

```
                {
                    break;
                }
            }
            return _filterPosition == _patternSequence.Count;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance full 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 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)]
            public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
            ↪  sequencesToMatch)
            {
                foreach (var sequenceToMatch in sequencesToMatch)
                {
                    if (PartialMatch(sequenceToMatch))
                    {
                        _readAsElements.Add(sequenceToMatch);
                        _results.Add(sequenceToMatch);
                    }
                }
            }
        }

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the sequences extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class SequencesExtensions
    {
        /// <summary>
        /// <para>
        /// Creates the sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="sequences">
        /// <para>The sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="groupedSequence">
        /// <para>The grouped sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
        ↪  groupedSequence)
        {
            var finalSequence = new TLink[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
```

```
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }

        /// <summary>
        /// <para>
        /// Returns the list using the specified sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="sequences">
        /// <para>The sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The list.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
        {
            var list = new List<TLink>();
            var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
            sequences.Each(filler.AddSkipFirstAndReturnConstant, new
            ↪  LinkAddress<TLink>(sequence));
            return list;
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.CriterionMatchers;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the sequences options.
    /// </para>
    /// <para></para>
    /// </summary>
    public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
    ↪  ILinks<TLink> must contain GetConstants function.
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        /// <summary>
        /// <para>
        /// Gets or sets the sequence marker link value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLink SequenceMarkerLink
        {
```

```
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <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
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

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

        // TODO: Реализовать компактификацию при чтении
        //public bool EnforceSingleSequenceVersionOnRead { get; set; }
        //public bool UseRequestMarker { get; set; }
        //public bool StoreRequestResults { get; set; }

        /// <summary>
        /// <para>
        /// Inits the options using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Cannot recreate sequence marker link.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void InitOptions(ISynchronizedLinks<TLink> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
                        if (!_equalityComparer.Equals(link, SequenceMarkerLink))
                        {
                            throw new InvalidOperationException("Cannot recreate sequence marker
                            ↪  link.");
                        }
                    }
                }
                if (MarkedSequenceMatcher == null)
                {
                    MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                    ↪  SequenceMarkerLink);
                }
            }
            var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
            if (UseCompression)
            {
                if (LinksToSequenceConverter == null)
```

```
                {
                    ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
                    if (UseSequenceMarker)
                    {
                        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>
    {
        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>
    {
        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>
```

```
41          /// <para></para>
42          /// </param>
43          /// <returns>
44          /// <para>The date time</para>
45          /// <para></para>
46          /// </returns>
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public DateTime Convert(TLink source) =>
            ↪  DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
49      }
50  }
```

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

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Singletons;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      /// <summary>
13      /// <para>
14      /// Represents the int 64 links extensions.
15      /// </para>
16      /// <para></para>
17      /// </summary>
18      public static class UInt64LinksExtensions
19      {
20          /// <summary>
21          /// <para>
22          /// Uses the unicode using the specified links.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="links">
27          /// <para>The links.</para>
28          /// <para></para>
29          /// </param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
32      }
33  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Converters;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Unicode
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the char to unicode symbol converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="LinksOperatorBase{TLink}"/>
15      /// <seealso cref="IConverter{char, TLink}"/>
16      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<char, TLink>
17      {
18          private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
            ↪  UncheckedConverter<char, TLink>.Default;
19
20          private readonly IConverter<TLink> _addressToNumberConverter;
21          private readonly TLink _unicodeSymbolMarker;
22
23          /// <summary>
24          /// <para>
25          /// Initializes a new <see cref="CharToUnicodeSymbolConverter"/> instance.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          /// <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>
    {
        private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
        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)]
        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
        ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
        ↪  unicodeSequenceMarker)
            : this(links, stringToUnicodeSymbolListConverter, new
            ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
            ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }

        /// <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="charToUnicodeSymbolConverter">
        /// <para>A char to unicode symbol 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)]
```

```
105         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
            ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
106             : this(links, new
                ↪ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
                ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
107
108         /// <summary>
109         /// <para>
110         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="links">
115         /// <para>A links.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="charToUnicodeSymbolConverter">
119         /// <para>A char to unicode symbol converter.</para>
120         /// <para></para>
121         /// </param>
122         /// <param name="listToSequenceLinkConverter">
123         /// <para>A list to sequence link converter.</para>
124         /// <para></para>
125         /// </param>
126         /// <param name="unicodeSequenceMarker">
127         /// <para>A unicode sequence marker.</para>
128         /// <para></para>
129         /// </param>
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
            ↪ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
            ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
132             : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
                ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
133
134         /// <summary>
135         /// <para>
136         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
137         /// </para>
138         /// <para></para>
139         /// </summary>
140         /// <param name="links">
141         /// <para>A links.</para>
142         /// <para></para>
143         /// </param>
144         /// <param name="stringToUnicodeSymbolListConverter">
145         /// <para>A string to unicode symbol list converter.</para>
146         /// <para></para>
147         /// </param>
148         /// <param name="listToSequenceLinkConverter">
149         /// <para>A list to sequence link converter.</para>
150         /// <para></para>
151         /// </param>
152         /// <param name="unicodeSequenceMarker">
153         /// <para>A unicode sequence marker.</para>
154         /// <para></para>
155         /// </param>
156         [MethodImpl(MethodImplOptions.AggressiveInlining)]
157         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
            ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
158             : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
                ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
159
160         /// <summary>
161         /// <para>
162         /// Converts the source.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="source">
167         /// <para>The source.</para>
168         /// <para></para>
169         /// </param>
170         /// <returns>
171         /// <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>>
    {
        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;

        private readonly ILinks<ulong> _links;
        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>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromLinksToString(IList<ulong> linksList)
        {
            var sb = new StringBuilder();
            for (int i = 0; i < linksList.Count; i++)
            {
                sb.Append(FromLinkToChar(linksList[i]));
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Creates the sequence link to string using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
            var sb = new StringBuilder();
            if (links.Exists(link))
            {
                StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                    x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                    ↪  element =>
                    {
                        sb.Append(FromLinkToChar(element));
                        return true;
                    });
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Creates the chars to link array using the specified chars.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
        ↪  chars.Length);

        /// <summary>
        /// <para>
        /// Creates the chars to link array using the specified chars.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <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>
    {
        private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLink> _sequenceWalker;
        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>
    {
        private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
        ↪  UncheckedConverter<TLink, char>.Default;

        private readonly IConverter<TLink> _numberToAddressConverter;
        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>
    {
        private readonly ISequenceIndex<TLink> _index;
        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
        private readonly TLink _unicodeSequenceMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</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)]
        public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
        ↪  ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
        ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
        {
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
        ↪  instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</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 UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
        ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
        ↪  unicodeSequenceMarker)
            : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
            ↪  unicodeSequenceMarker) { }

        /// <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>
        /// <para>
        /// Initializes a new <see cref="LeftSequenceWalker"/> 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 LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
        ↪  isElement) : base(links, stack, isElement) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeftSequenceWalker"/> 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 LeftSequenceWalker(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.GetSource(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.GetTarget(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 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>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        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>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
        ↪  base(links) => _isElement = isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
        ↪  _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) => ToArray(sequence);

        /// <summary>
        /// <para>
        /// Returns the array using the specified sequence.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    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>
    {
        private readonly IStack<TLink> _stack;
        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>
```

```
123         /// </param>
124         /// <returns>
125         /// <para>The bool</para>
126         /// <para></para>
127         /// </returns>
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
130 
131         /// <summary>
132         /// <para>
133         /// Gets the next element after pop using the specified element.
134         /// </para>
135         /// <para></para>
136         /// </summary>
137         /// <param name="element">
138         /// <para>The element.</para>
139         /// <para></para>
140         /// </param>
141         /// <returns>
142         /// <para>The link</para>
143         /// <para></para>
144         /// </returns>
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         protected abstract TLink GetNextElementAfterPop(TLink element);
147 
148         /// <summary>
149         /// <para>
150         /// Gets the next element after push using the specified element.
151         /// </para>
152         /// <para></para>
153         /// </summary>
154         /// <param name="element">
155         /// <para>The element.</para>
156         /// <para></para>
157         /// </param>
158         /// <returns>
159         /// <para>The link</para>
160         /// <para></para>
161         /// </returns>
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         protected abstract TLink GetNextElementAfterPush(TLink element);
164 
165         /// <summary>
166         /// <para>
167         /// Walks the contents using the specified element.
168         /// </para>
169         /// <para></para>
170         /// </summary>
171         /// <param name="element">
172         /// <para>The element.</para>
173         /// <para></para>
174         /// </param>
175         /// <returns>
176         /// <para>An enumerable of t link</para>
177         /// <para></para>
178         /// </returns>
179         [MethodImpl(MethodImplOptions.AggressiveInlining)]
180         protected abstract IEnumerable<TLink> WalkContents(TLink element);
181     }
182 }
```

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>
39          /// </summary>
40          /// <typeparam name="TLink">
41          /// <para>The link.</para>
42          /// <para></para>
43          /// </typeparam>
44          /// <param name="dataDbFilename">
45          /// <para>The data db filename.</para>
46          /// <para></para>
47          /// </param>
48          /// <returns>
49          /// <para>A links of t link</para>
50          /// <para></para>
51          /// </returns>
52          public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
53          {
54              var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
                ↪  true);
55              return new UnitedMemoryLinks<TLink>(new
                ↪  FileMappedResizableDirectMemory(dataDbFilename),
                ↪  UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
56          }
57
58          /// <summary>
59          /// <para>
60          /// Tests that decimal max value test.
61          /// </para>
62          /// <para></para>
63          /// </summary>
64          [Fact]
65          public void DecimalMaxValueTest()
66          {
67              var links = CreateLinks();
68              BigInteger bigInteger = new(decimal.MaxValue);
69              TLink negativeNumberMarker = links.Create();
70              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
71              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
72              BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
73              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
                ↪  negativeNumberMarker);
74              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
75              var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
76              var bigIntFromSequence =
                ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
77              Assert.Equal(bigInteger, bigIntFromSequence);
78          }
79
80          /// <summary>
81          /// <para>
82          /// Tests that decimal min value test.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          [Fact]
87          public void DecimalMinValueTest()
```

```
88          {
89              var links = CreateLinks();
90              BigInteger bigInteger = new(decimal.MinValue);
91              TLink negativeNumberMarker = links.Create();
92              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
93              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
94              BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
95              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
                ↪  negativeNumberMarker);
96              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
97              var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
98              var bigIntFromSequence =
                ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
99              Assert.Equal(bigInteger, bigIntFromSequence);
100         }
101
102         /// <summary>
103         /// <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;
```

```
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Numbers.Raw;
using Platform.Interfaces;
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 default sequence appender tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class DefaultSequenceAppenderTests
    {
        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
    {
        private static readonly string _sequenceExample = "зеленела зелёная зелень";
        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);
```

```
112                 var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
113                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
114                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
115
116                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
                    ↪  new LeveledSequenceWalker<ulong>(links) });
117
118                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
119             }
120         }
121
122         /// <summary>
123         /// <para>
124         /// Tests that dictionary based frequency stored optimal variant sequence test.
125         /// </para>
126         /// <para></para>
127         /// </summary>
128         [Fact]
129         public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
130         {
131             using (var scope = new TempLinksTestScope(useSequences: false))
132             {
133                 var links = scope.Links;
134
135                 links.UseUnicode();
136
137                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
138
139                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
140
141                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
142
143                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
144                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
145
146                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
147                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
148
149                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
                    ↪  new LeveledSequenceWalker<ulong>(links) });
150
151                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                    ↪  index, optimalVariantConverter);
152             }
153         }
154
155         private static void ExecuteTest(Sequences sequences, ulong[] sequence,
            ↪  SequenceToItsLocalElementLevelsConverter<ulong>
            ↪  sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
            ↪  OptimalVariantConverter<ulong> optimalVariantConverter)
156         {
157             index.Add(sequence);
158
159             var optimalVariant = optimalVariantConverter.Convert(sequence);
160
161             var readSequence1 = sequences.ToList(optimalVariant);
162
163             Assert.True(sequence.SequenceEqual(readSequence1));
164         }
165
166         /// <summary>
167         /// <para>
168         /// Tests that saved sequences optimization test.
169         /// </para>
170         /// <para></para>
```

```
171         /// </summary>
172         [Fact]
173         public static void SavedSequencesOptimizationTest()
174         {
175             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
                ↪  (long.MaxValue + 1UL, ulong.MaxValue));
176 
177             using (var memory = new HeapResizableDirectMemory())
178             using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
                ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
179             {
180                 var links = new UInt64Links(disposableLinks);
181 
182                 var root = links.CreatePoint();
183 
184                 //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
185                 var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
186 
187                 var unicodeSymbolMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(1));
188                 var unicodeSequenceMarker = links.GetOrCreate(root,
                    ↪  addressToNumberConverter.Convert(2));
189 
190                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
191                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                    ↪  totalSequenceSymbolFrequencyCounter);
192                 var index = new
                    ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
193                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);
194                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
195                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
196 
197                 var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
                    ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
198 
199                 var unicodeSequencesOptions = new SequencesOptions<ulong>()
200                 {
201                     UseSequenceMarker = true,
202                     SequenceMarkerLink = unicodeSequenceMarker,
203                     UseIndex = true,
204                     Index = index,
205                     LinksToSequenceConverter = optimalVariantConverter,
206                     Walker = walker,
207                     UseGarbageCollection = true
208                 };
209 
210                 var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
                    ↪  unicodeSequencesOptions);
211 
212                 // Create some sequences
213                 var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
                    ↪  StringSplitOptions.RemoveEmptyEntries);
214                 var arrays = strings.Select(x => x.Select(y =>
                    ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
215                 for (int i = 0; i < arrays.Length; i++)
216                 {
217                     unicodeSequences.Create(arrays[i].ShiftRight());
218                 }
219 
220                 var linksCountAfterCreation = links.Count();
221 
222                 // get list of sequences links
223                 // for each sequence link
224                 //   create new sequence version
225                 //   if new sequence is not the same as sequence link
226                 //     delete sequence link
227                 //     collect garbadge
228                 unicodeSequences.CompactAll();
229 
230                 var linksCountAfterCompactification = links.Count();
231 
232                 Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
233             }
```

```
234             }
235         }
236     }
```

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

```
1   using Platform.Data.Doublets.Memory;
2   using Platform.Data.Doublets.Memory.United.Generic;
3   using Platform.Data.Doublets.Numbers.Rational;
4   using Platform.Data.Doublets.Numbers.Raw;
5   using Platform.Data.Doublets.Sequences.Converters;
6   using Platform.Data.Numbers.Raw;
7   using Platform.Memory;
8   using Xunit;
9   using TLink = System.UInt64;
10
11  namespace Platform.Data.Doublets.Sequences.Tests
12  {
13      /// <summary>
14      /// <para>
15      /// Represents the rational numbers tests.
16      /// </para>
17      /// <para></para>
18      /// </summary>
19      public class RationalNumbersTests
20      {
21          /// <summary>
22          /// <para>
23          /// Creates the links.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          /// <returns>
28          /// <para>A links of t link</para>
29          /// <para></para>
30          /// </returns>
31          public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
32
33          /// <summary>
34          /// <para>
35          /// Creates the links using the specified data db filename.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <typeparam name="TLink">
40          /// <para>The link.</para>
41          /// <para></para>
42          /// </typeparam>
43          /// <param name="dataDbFilename">
44          /// <para>The data db filename.</para>
45          /// <para></para>
46          /// </param>
47          /// <returns>
48          /// <para>A links of t link</para>
49          /// <para></para>
50          /// </returns>
51          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);
```

```
            var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
            Assert.Equal(@decimal, decimalFromRational);
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Xunit;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Sequences;

namespace Platform.Data.Doublets.Sequences.Tests
{
    /// <summary>
    /// <para>
    /// Represents the read sequence tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ReadSequenceTests
    {
        /// <summary>
        /// <para>
        /// Tests that read sequence test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ReadSequenceTest()
        {
            const long sequenceLength = 2000;

            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;
                var sequences = new Sequences(links, new SequencesOptions<ulong> { Walker = new
                ↪  LeveledSequenceWalker<ulong>(links) });

                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 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
    {
        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]);
            }
        }
    }
```

```
231
232         /// <summary>
233         /// <para>
234         /// Tests that all partial variants search test.
235         /// </para>
236         /// <para></para>
237         /// </summary>
238         [Fact]
239         public static void AllPartialVariantsSearchTest()
240         {
241             const long sequenceLength = 8;
242
243             using (var scope = new TempLinksTestScope(useSequences: true))
244             {
245                 var links = scope.Links;
246                 var sequences = scope.Sequences;
247
248                 var sequence = new ulong[sequenceLength];
249                 for (var i = 0; i < sequenceLength; i++)
250                 {
251                     sequence[i] = links.Create();
252                 }
253
254                 var createResults = sequences.CreateAllVariants2(sequence);
255
256                 //var createResultsStrings = createResults.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
257                 //Global.Trash = createResultsStrings;
258
259                 var partialSequence = new ulong[sequenceLength - 2];
260
261                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
262
263                 var sw1 = Stopwatch.StartNew();
264                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
265
266                 var sw2 = Stopwatch.StartNew();
267                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
268
269                 //var sw3 = Stopwatch.StartNew();
270                 //var searchResults3 =
                    ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
271
272                 var sw4 = Stopwatch.StartNew();
273                 var searchResults4 =
                    ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
274
275                 //Global.Trash = searchResults3;
276
277                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
                    ↪ sequences.FormatSequence(x)).ToList();
278                 //Global.Trash = searchResults1Strings;
279
280                 var intersection1 = createResults.Intersect(searchResults1).ToList();
281                 Assert.True(intersection1.Count == createResults.Length);
282
283                 var intersection2 = createResults.Intersect(searchResults2).ToList();
284                 Assert.True(intersection2.Count == createResults.Length);
285
286                 var intersection4 = createResults.Intersect(searchResults4).ToList();
287                 Assert.True(intersection4.Count == createResults.Length);
288
289                 for (var i = 0; i < sequenceLength; i++)
290                 {
291                     links.Delete(sequence[i]);
292                 }
293             }
294         }
295
296         /// <summary>
297         /// <para>
298         /// Tests that balanced partial variants search test.
299         /// </para>
300         /// <para></para>
301         /// </summary>
302         [Fact]
303         public static void BalancedPartialVariantsSearchTest()
```

```
304         {
305             const long sequenceLength = 200;
306
307             using (var scope = new TempLinksTestScope(useSequences: true))
308             {
309                 var links = scope.Links;
310                 var sequences = scope.Sequences;
311
312                 var sequence = new ulong[sequenceLength];
313                 for (var i = 0; i < sequenceLength; i++)
314                 {
315                     sequence[i] = links.Create();
316                 }
317
318                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
319
320                 var balancedVariant = balancedVariantConverter.Convert(sequence);
321
322                 var partialSequence = new ulong[sequenceLength - 2];
323
324                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
325
326                 var sw1 = Stopwatch.StartNew();
327                 var searchResults1 =
                    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
328
329                 var sw2 = Stopwatch.StartNew();
330                 var searchResults2 =
                    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
331
332                 Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
333
334                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
                    ↪ searchResults2.First());
335
336                 for (var i = 0; i < sequenceLength; i++)
337                 {
338                     links.Delete(sequence[i]);
339                 }
340             }
341         }
342
343         /// <summary>
344         /// <para>
345         /// Tests that pattern match test.
346         /// </para>
347         /// <para></para>
348         /// </summary>
349         [Fact(Skip = "Correct implementation is pending")]
350         public static void PatternMatchTest()
351         {
352             var zeroOrMany = Sequences.ZeroOrMany;
353
354             using (var scope = new TempLinksTestScope(useSequences: true))
355             {
356                 var links = scope.Links;
357                 var sequences = scope.Sequences;
358
359                 var e1 = links.Create();
360                 var e2 = links.Create();
361
362                 var sequence = new[]
363                 {
364                     e1, e2, e1, e2 // mama / papa
365                 };
366
367                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
368
369                 var balancedVariant = balancedVariantConverter.Convert(sequence);
370
371                 // 1: [1]
372                 // 2: [2]
373                 // 3: [1,2]
374                 // 4: [1,2,1,2]
375
376                 var doublet = links.GetSource(balancedVariant);
377
378                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
379
380                 Assert.True(matchedSequences1.Count == 0);
```

```
                var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);

                Assert.True(matchedSequences2.Count == 0);

                var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);

                Assert.True(matchedSequences3.Count == 0);

                var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);

                Assert.Contains(doublet, matchedSequences4);
                Assert.Contains(balancedVariant, matchedSequences4);

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

        /// <summary>
        /// <para>
        /// Tests that index test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void IndexTest()
        {
            using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
             ↪  true }, useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;
                var index = sequences.Options.Index;

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa
                };

                Assert.False(index.MightContain(sequence));

                index.Add(sequence);

                Assert.True(index.MightContain(sequence));
            }
        }

        private static readonly string _exampleText =
            @"([english
             ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

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?

[![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)

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?

[![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)

А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
↪  так? Инверсия? Отражение? Сумма?
```

[![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)

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?

[![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)

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?

[![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)

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?

[![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)

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?

[![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)

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)";

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

                var e1 = links.Create();
                var e2 = links.Create();

                var sequence = new[]
                {
                    e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
                };

                var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
                var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
                var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync, totalSequenceSymbolFrequencyCounter);
                var compressingConverter = new CompressingConverter<ulong>(links.Unsync, balancedVariantConverter, doubletFrequenciesCache);

                var compressedVariant = compressingConverter.Convert(sequence);

                // 1: [1]       (1->1) point
                // 2: [2]       (2->2) point
                // 3: [1,2]     (1->2) doublet
                // 4: [1,2,1,2] (3->3) doublet

                Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
                Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
                Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
                Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);

                var source = _constants.SourcePart;
                var target = _constants.TargetPart;

                Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
                Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
                Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
                Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
```

```
538
539                 // 4 - length of sequence
540                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
                    ↪  == sequence[0]);
541                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
                    ↪  == sequence[1]);
542                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
                    ↪  == sequence[2]);
543                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
                    ↪  == sequence[3]);
544             }
545         }
546
547         /// <summary>
548         /// <para>
549         /// Tests that compression efficiency test.
550         /// </para>
551         /// <para></para>
552         /// </summary>
553         [Fact]
554         public static void CompressionEfficiencyTest()
555         {
556             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
                    ↪  StringSplitOptions.RemoveEmptyEntries);
557             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
558             var totalCharacters = arrays.Select(x => x.Length).Sum();
559
560             using (var scope1 = new TempLinksTestScope(useSequences: true))
561             using (var scope2 = new TempLinksTestScope(useSequences: true))
562             using (var scope3 = new TempLinksTestScope(useSequences: true))
563             {
564                 scope1.Links.Unsync.UseUnicode();
565                 scope2.Links.Unsync.UseUnicode();
566                 scope3.Links.Unsync.UseUnicode();
567
568                 var balancedVariantConverter1 = new
                    ↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
569                 var totalSequenceSymbolFrequencyCounter = new
                    ↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
570                 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
571                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
                    ↪  balancedVariantConverter1, linkFrequenciesCache1,
                    ↪  doInitialFrequenciesIncrement: false);
572
573                 //var compressor2 = scope2.Sequences;
574                 var compressor3 = scope3.Sequences;
575
576                 var constants = Default<LinksConstants<ulong>>.Instance;
577
578                 var sequences = compressor3;
579                 //var meaningRoot = links.CreatePoint();
580                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
581                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
582                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
                    ↪  constants.Itself);
583
584                 //var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
585                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
                    ↪  unaryOne);
586                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
587                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
588                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
589                 //var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
590
591                 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
                    ↪  totalSequenceSymbolFrequencyCounter);
592
593                 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<ulong>(linkFrequenciesCache3);
594
```

```
595                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
                    ↪  linkToItsFrequencyNumberConverter);
596                 var optimalVariantConverter = new
                    ↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
                    ↪  sequenceToItsLocalElementLevelsConverter);
597
598                 var compressed1 = new ulong[arrays.Length];
599                 var compressed2 = new ulong[arrays.Length];
600                 var compressed3 = new ulong[arrays.Length];
601
602                 var START = 0;
603                 var END = arrays.Length;
604
605                 //for (int i = START; i < END; i++)
606                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
607
608                 var initialCount1 = scope2.Links.Unsync.Count();
609
610                 var sw1 = Stopwatch.StartNew();
611
612                 for (int i = START; i < END; i++)
613                 {
614                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
615                     compressed1[i] = compressor1.Convert(arrays[i]);
616                 }
617
618                 var elapsed1 = sw1.Elapsed;
619
620                 var balancedVariantConverter2 = new
                    ↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
621
622                 var initialCount2 = scope2.Links.Unsync.Count();
623
624                 var sw2 = Stopwatch.StartNew();
625
626                 for (int i = START; i < END; i++)
627                 {
628                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
629                 }
630
631                 var elapsed2 = sw2.Elapsed;
632
633                 for (int i = START; i < END; i++)
634                 {
635                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
636                 }
637
638                 var initialCount3 = scope3.Links.Unsync.Count();
639
640                 var sw3 = Stopwatch.StartNew();
641
642                 for (int i = START; i < END; i++)
643                 {
644                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
645                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
646                 }
647
648                 var elapsed3 = sw3.Elapsed;
649
650                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
                    ↪  Optimal variant: {elapsed3}");
651
652                 // Assert.True(elapsed1 > elapsed2);
653
654                 // Checks
655                 for (int i = START; i < END; i++)
656                 {
657                     var sequence1 = compressed1[i];
658                     var sequence2 = compressed2[i];
659                     var sequence3 = compressed3[i];
660
661                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                        ↪  scope1.Links.Unsync);
662
663                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                        ↪  scope2.Links.Unsync);
664
665                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
                        ↪  scope3.Links.Unsync);
```

```
666
667                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
                        ↪  link.IsPartialPoint());
668                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
                        ↪  link.IsPartialPoint());
669                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
                        ↪  link.IsPartialPoint());
670
671                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
672                     //    Assert.False(structure1 == structure2);
673                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
                        ↪  arrays[i].Length > 3)
674                     //    Assert.False(structure3 == structure2);
675
676                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
677                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
678                 }
679
680                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
                    ↪  totalCharacters);
681                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
                    ↪  totalCharacters);
682                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
                    ↪  totalCharacters);
683
684                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
                    ↪  totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
                    ↪  totalCharacters}");
685
686                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
687                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
                    ↪  scope2.Links.Unsync.Count() - initialCount2);
688
689                 var duplicateProvider1 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
690                 var duplicateProvider2 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
691                 var duplicateProvider3 = new
                    ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
692
693                 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
694                 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
695                 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
696
697                 var duplicates1 = duplicateCounter1.Count();
698
699                 ConsoleHelpers.Debug("------");
700
701                 var duplicates2 = duplicateCounter2.Count();
702
703                 ConsoleHelpers.Debug("------");
704
705                 var duplicates3 = duplicateCounter3.Count();
706
707                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
708
709                 linkFrequenciesCache1.ValidateFrequencies();
710                 linkFrequenciesCache3.ValidateFrequencies();
711             }
712         }
713
714         /// <summary>
715         /// <para>
716         /// Tests that compression stability test.
717         /// </para>
718         /// <para></para>
719         /// </summary>
720         [Fact]
721         public static void CompressionStabilityTest()
722         {
723             // TODO: Fix bug (do a separate test)
724             //const ulong minNumbers = 0;
725             //const ulong maxNumbers = 1000;
726
727             const ulong minNumbers = 10000;
```

```
728             const ulong maxNumbers = 12500;
729
730             var strings = new List<string>();
731
732             for (ulong i = minNumbers; i < maxNumbers; i++)
733             {
734                 strings.Add(i.ToString());
735             }
736
737             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
738             var totalCharacters = arrays.Select(x => x.Length).Sum();
739
740             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↪  SequencesOptions<ulong> { UseCompression = true,
    ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
741             using (var scope2 = new TempLinksTestScope(useSequences: true))
742             {
743                 scope1.Links.UseUnicode();
744                 scope2.Links.UseUnicode();
745
746                 //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
747                 var compressor1 = scope1.Sequences;
748                 var compressor2 = scope2.Sequences;
749
750                 var compressed1 = new ulong[arrays.Length];
751                 var compressed2 = new ulong[arrays.Length];
752
753                 var sw1 = Stopwatch.StartNew();
754
755                 var START = 0;
756                 var END = arrays.Length;
757
758                 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
759                 // Stability issue starts at 10001 or 11000
760                 //for (int i = START; i < END; i++)
761                 //{
762                 //    var first = compressor1.Compress(arrays[i]);
763                 //    var second = compressor1.Compress(arrays[i]);
764
765                 //    if (first == second)
766                 //        compressed1[i] = first;
767                 //    else
768                 //    {
769                 //        // TODO: Find a solution for this case
770                 //    }
771                 //}
772
773                 for (int i = START; i < END; i++)
774                 {
775                     var first = compressor1.Create(arrays[i].ShiftRight());
776                     var second = compressor1.Create(arrays[i].ShiftRight());
777
778                     if (first == second)
779                     {
780                         compressed1[i] = first;
781                     }
782                     else
783                     {
784                         // TODO: Find a solution for this case
785                     }
786                 }
787
788                 var elapsed1 = sw1.Elapsed;
789
790                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
791
792                 var sw2 = Stopwatch.StartNew();
793
794                 for (int i = START; i < END; i++)
795                 {
796                     var first = balancedVariantConverter.Convert(arrays[i]);
797                     var second = balancedVariantConverter.Convert(arrays[i]);
798
799                     if (first == second)
800                     {
801                         compressed2[i] = first;
802                     }
803                 }
804
805                 var elapsed2 = sw2.Elapsed;
```

```
806
807                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
808
809                 Assert.True(elapsed1 > elapsed2);
810
811                 // Checks
812                 for (int i = START; i < END; i++)
813                 {
814                     var sequence1 = compressed1[i];
815                     var sequence2 = compressed2[i];
816
817                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
818                     {
819                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
820
821                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
822
823                         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                            ↪  link.IsPartialPoint());
824                         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                            ↪  link.IsPartialPoint());
825
826                         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                            ↪  arrays[i].Length > 3)
827                         //    Assert.False(structure1 == structure2);
828
829                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
830                     }
831                 }
832
833                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
834                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
835
836                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
837
838                 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
839
840                 //compressor1.ValidateFrequencies();
841             }
842         }
843
844         /// <summary>
845         /// <para>
846         /// Tests that rundom numbers compression quality test.
847         /// </para>
848         /// <para></para>
849         /// </summary>
850         [Fact]
851         public static void RundomNumbersCompressionQualityTest()
852         {
853             const ulong N = 500;
854
855             //const ulong minNumbers = 10000;
856             //const ulong maxNumbers = 20000;
857
858             //var strings = new List<string>();
859
860             //for (ulong i = 0; i < N; i++)
861             //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
                    ↪  maxNumbers).ToString());
862
863             var strings = new List<string>();
864
865             for (ulong i = 0; i < N; i++)
866             {
867                 strings.Add(RandomHelpers.Default.NextUInt64().ToString());
868             }
869
870             strings = strings.Distinct().ToList();
871
872             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
873             var totalCharacters = arrays.Select(x => x.Length).Sum();
874
```

```
875             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
                ↪  SequencesOptions<ulong> { UseCompression = true,
                ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
876             using (var scope2 = new TempLinksTestScope(useSequences: true))
877             {
878                 scope1.Links.UseUnicode();
879                 scope2.Links.UseUnicode();
880
881                 var compressor1 = scope1.Sequences;
882                 var compressor2 = scope2.Sequences;
883
884                 var compressed1 = new ulong[arrays.Length];
885                 var compressed2 = new ulong[arrays.Length];
886
887                 var sw1 = Stopwatch.StartNew();
888
889                 var START = 0;
890                 var END = arrays.Length;
891
892                 for (int i = START; i < END; i++)
893                 {
894                     compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
895                 }
896
897                 var elapsed1 = sw1.Elapsed;
898
899                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
900
901                 var sw2 = Stopwatch.StartNew();
902
903                 for (int i = START; i < END; i++)
904                 {
905                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
906                 }
907
908                 var elapsed2 = sw2.Elapsed;
909
910                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
                    ↪  {elapsed2}");
911
912                 Assert.True(elapsed1 > elapsed2);
913
914                 // Checks
915                 for (int i = START; i < END; i++)
916                 {
917                     var sequence1 = compressed1[i];
918                     var sequence2 = compressed2[i];
919
920                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
921                     {
922                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                            ↪  scope1.Links);
923
924                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                            ↪  scope2.Links);
925
926                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
927                     }
928                 }
929
930                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
931                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
932
933                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
                    ↪  totalCharacters}");
934
935                 // Can be worse than balanced variant
936                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
937
938                 //compressor1.ValidateFrequencies();
939             }
940         }
941
942         /// <summary>
943         /// <para>
944         /// Tests that all tree break down at sequences creation bug test.
945         /// </para>
946         /// <para></para>
```

```
        /// </summary>
        [Fact]
        public static void AllTreeBreakDownAtSequencesCreationBugTest()
        {
            // Made out of AllPossibleConnectionsTest test.

            //const long sequenceLength = 5; //100% bug
            const long sequenceLength = 4; //100% bug
            //const long sequenceLength = 3; //100% _no_bug (ok)

            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);

                Global.Trash = createResults;

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

        /// <summary>
        /// <para>
        /// Tests that all possible connections test.
        /// </para>
        /// <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>
```

```
13      /// Represents the temp links test scope.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      /// <seealso cref="DisposableBase"/>
18      public class TempLinksTestScope : DisposableBase
19      {
20          /// <summary>
21          /// <para>
22          /// Gets the memory adapter value.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          public ILinks<ulong> MemoryAdapter { get; }
27          /// <summary>
28          /// <para>
29          /// Gets the links value.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          public SynchronizedLinks<ulong> Links { get; }
34          /// <summary>
35          /// <para>
36          /// Gets the sequences value.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          public Sequences Sequences { get; }
41          /// <summary>
42          /// <para>
43          /// Gets the temp filename value.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          public string TempFilename { get; }
48          /// <summary>
49          /// <para>
50          /// Gets the temp transaction log filename value.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          public string TempTransactionLogFilename { get; }
55          private readonly bool _deleteFiles;
56
57          /// <summary>
58          /// <para>
59          /// Initializes a new <see cref="TempLinksTestScope"/> instance.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="deleteFiles">
64          /// <para>A delete files.</para>
65          /// <para></para>
66          /// </param>
67          /// <param name="useSequences">
68          /// <para>A use sequences.</para>
69          /// <para></para>
70          /// </param>
71          /// <param name="useLog">
72          /// <para>A use log.</para>
73          /// <para></para>
74          /// </param>
75          public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
            ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
            ↪  useLog) { }
76
77          /// <summary>
78          /// <para>
79          /// Initializes a new <see cref="TempLinksTestScope"/> instance.
80          /// </para>
81          /// <para></para>
82          /// </summary>
83          /// <param name="sequencesOptions">
84          /// <para>A sequences options.</para>
85          /// <para></para>
86          /// </param>
87          /// <param name="deleteFiles">
88          /// <para>A delete files.</para>
```

```
89          /// <para></para>
90          /// </param>
91          /// <param name="useSequences">
92          /// <para>A use sequences.</para>
93          /// <para></para>
94          /// </param>
95          /// <param name="useLog">
96          /// <para>A use log.</para>
97          /// <para></para>
98          /// </param>
99          public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
            ↪ true, bool useSequences = false, bool useLog = false)
100         {
101             _deleteFiles = deleteFiles;
102             TempFilename = Path.GetTempFileName();
103             TempTransactionLogFilename = Path.GetTempFileName();
104             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
105             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
                ↪ FileMappedResizableDirectMemory(TempFilename), new
                ↪ FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
                ↪ UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
                ↪ Memory.IndexTreeType.Default, useLinkedList: true);
106             MemoryAdapter = useLog ? (ILinks<ulong>)new
                ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
                ↪ coreMemoryAdapter;
107             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
108             if (useSequences)
109             {
110                 Sequences = new Sequences(Links, sequencesOptions);
111             }
112         }
113
114         /// <summary>
115         /// <para>
116         /// Disposes the manual.
117         /// </para>
118         /// <para></para>
119         /// </summary>
120         /// <param name="manual">
121         /// <para>The manual.</para>
122         /// <para></para>
123         /// </param>
124         /// <param name="wasDisposed">
125         /// <para>The was disposed.</para>
126         /// <para></para>
127         /// </param>
128         protected override void Dispose(bool manual, bool wasDisposed)
129         {
130             if (!wasDisposed)
131             {
132                 Links.Unsync.DisposeIfPossible();
133                 if (_deleteFiles)
134                 {
135                     DeleteFiles();
136                 }
137             }
138         }
139
140         /// <summary>
141         /// <para>
142         /// Deletes the files.
143         /// </para>
144         /// <para></para>
145         /// </summary>
146         public void DeleteFiles()
147         {
148             File.Delete(TempFilename);
149             File.Delete(TempTransactionLogFilename);
150         }
151     }
152 }
```

1.69 ./csharp/Platform.Data.Doublets.Sequences.Tests/TestExtensions.cs

```
1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  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;

            var equalityComparer = EqualityComparer<T>.Default;

            var zero = default(T);
            var one = Arithmetic.Increment(zero);

            // Create Link
            Assert.True(equalityComparer.Equals(links.Count(), zero));

            var setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));

            var linkAddress = links.Create();

            var link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(link.Count == 3);
            Assert.True(equalityComparer.Equals(link.Index, linkAddress));
            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            Assert.True(equalityComparer.Equals(links.Count(), one));

            // Get first link
            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);

            Assert.True(equalityComparer.Equals(setter.Result, linkAddress));

            // Update link to reference itself
            links.Update(linkAddress, linkAddress, linkAddress);

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, linkAddress));
            Assert.True(equalityComparer.Equals(link.Target, linkAddress));

            // Update link to reference null (prepare for delete)
            var updated = links.Update(linkAddress, constants.Null, constants.Null);

            Assert.True(equalityComparer.Equals(updated, linkAddress));

            link = new Link<T>(links.GetLink(linkAddress));

            Assert.True(equalityComparer.Equals(link.Source, constants.Null));
            Assert.True(equalityComparer.Equals(link.Target, constants.Null));

            // Delete link
```

```
            links.Delete(linkAddress);

            Assert.True(equalityComparer.Equals(links.Count(), zero));

            setter = new Setter<T>(constants.Null);
            links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);

            Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
        }

        /// <summary>
        /// <para>
        /// Tests the raw numbers 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 TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
        {
            // Constants
            var constants = links.Constants;
            var equalityComparer = EqualityComparer<T>.Default;

            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>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximumOperationsPerCycle">
        /// <para>The maximum operations per cycle.</para>
        /// <para></para>
        /// </param>
        public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
        ↪  links, int maximumOperationsPerCycle)
        {
            var comparer = Comparer<TLink>.Default;
            var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
            var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var created = 0UL;
                var deleted = 0UL;
                for (var i = 0; i < N; i++)
                {
                    var linksCount = addressToUInt64Converter.Convert(links.Count());
                    var createPoint = random.NextBoolean();
                    if (linksCount >= 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, linksCount);
                        TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                        ↪  ddressRange));
                        ↪  //-V3086
                        var resultLink = links.GetOrCreate(source, target);
                        if (comparer.Compare(resultLink,
                        ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
                        {
                            created++;
                        }
                    }
                    else
                    {
                        links.Create();
                        created++;
                    }
                }
```

```
                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
                for (var i = 0; i < N; i++)
                {
                    TLink link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
                    if (links.Exists(link))
                    {
                        links.Delete(link);
                        deleted++;
                    }
                }
                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
            }
        }
    }
}
```

1.70 ./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
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
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪ Default<LinksConstants<ulong>>.Instance;

        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]
```

```
61          public static void CascadeUpdateTest()
62          {
63              var itself = _constants.Itself;
64              using (var scope = new TempLinksTestScope(useLog: true))
65              {
66                  var links = scope.Links;
67
68                  var l1 = links.Create();
69                  var l2 = links.Create();
70
71                  l2 = links.Update(l2, l2, l1, l2);
72
73                  links.CreateAndUpdate(l2, itself);
74                  links.CreateAndUpdate(l2, itself);
75
76                  l2 = links.Update(l2, l1);
77
78                  links.Delete(l2);
79
80                  Global.Trash = links.Count();
81
82                  links.Unsync.DisposeIfPossible(); // Close links to access log
83
84                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
85              }
86          }
87
88          /// <summary>
89          /// <para>
90          /// Tests that basic transaction log test.
91          /// </para>
92          /// <para></para>
93          /// </summary>
94          [Fact]
95          public static void BasicTransactionLogTest()
96          {
97              using (var scope = new TempLinksTestScope(useLog: true))
98              {
99                  var links = scope.Links;
100                 var l1 = links.Create();
101                 var l2 = links.Create();
102
103                 Global.Trash = links.Update(l2, l2, l1, l2);
104
105                 links.Delete(l1);
106
107                 links.Unsync.DisposeIfPossible(); // Close links to access log
108
109                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
110             }
111         }
112
113         /// <summary>
114         /// <para>
115         /// Tests that transaction auto reverted test.
116         /// </para>
117         /// <para></para>
118         /// </summary>
119         [Fact]
120         public static void TransactionAutoRevertedTest()
121         {
122             // Auto Reverted (Because no commit at transaction)
123             using (var scope = new TempLinksTestScope(useLog: true))
124             {
125                 var links = scope.Links;
126                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
127                 using (var transaction = transactionsLayer.BeginTransaction())
128                 {
129                     var l1 = links.Create();
130                     var l2 = links.Create();
131
132                     links.Update(l2, l2, l1, l2);
133                 }
134
135                 Assert.Equal(0UL, links.Count());
136
137                 links.Unsync.DisposeIfPossible();
```

```
138
139                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
                    ↪  cope.TempTransactionLogFilename);
140                 Assert.Single(transitions);
141             }
142         }
143
144         /// <summary>
145         /// <para>
146         /// Tests that transaction user code error no data saved test.
147         /// </para>
148         /// <para></para>
149         /// </summary>
150         [Fact]
151         public static void TransactionUserCodeErrorNoDataSavedTest()
152         {
153             // User Code Error (Autoreverted), no data saved
154             var itself = _constants.Itself;
155
156             TempLinksTestScope lastScope = null;
157             try
158             {
159                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                    ↪  useLog: true))
160                 {
161                     var links = scope.Links;
162                     var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
                        ↪  atorBase<ulong>)links.Unsync).Links;
163                     using (var transaction = transactionsLayer.BeginTransaction())
164                     {
165                         var l1 = links.CreateAndUpdate(itself, itself);
166                         var l2 = links.CreateAndUpdate(itself, itself);
167
168                         l2 = links.Update(l2, l2, l1, l2);
169
170                         links.CreateAndUpdate(l2, itself);
171                         links.CreateAndUpdate(l2, itself);
172
173                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
                            ↪  tion>(scope.TempTransactionLogFilename);
174
175                         l2 = links.Update(l2, l1);
176
177                         links.Delete(l2);
178
179                         ExceptionThrower();
180
181                         transaction.Commit();
182                     }
183
184                     Global.Trash = links.Count();
185                 }
186             }
187             catch
188             {
189                 Assert.False(lastScope == null);
190
191                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
                    ↪  astScope.TempTransactionLogFilename);
192
193                 Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
                    ↪  transitions[0].After.IsNull());
194
195                 lastScope.DeleteFiles();
196             }
197         }
198
199         /// <summary>
200         /// <para>
201         /// Tests that transaction user code error some data saved test.
202         /// </para>
203         /// <para></para>
204         /// </summary>
205         [Fact]
206         public static void TransactionUserCodeErrorSomeDataSavedTest()
207         {
208             // User Code Error (Autoreverted), some data saved
209             var itself = _constants.Itself;
210
```

```
            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();

                    Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
                    ↪  scope.TempTransactionLogFilename);
                }

                using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
                ↪  useLog: true))
                {
                    var links = scope.Links;
                    var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
                    using (var transaction = transactionsLayer.BeginTransaction())
                    {
                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Assert.False(lastScope == null);

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
                ↪  Scope.TempTransactionLogFilename);

                lastScope.DeleteFiles();
            }
        }

        /// <summary>
        /// <para>
        /// Tests that transaction commit.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void TransactionCommit()
        {
            var itself = _constants.Itself;

            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            // Commit
            using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
            using (var links = new UInt64Links(memoryAdapter))
            {
                using (var transaction = memoryAdapter.BeginTransaction())
                {
                    var l1 = links.CreateAndUpdate(itself, itself);
                    var l2 = links.CreateAndUpdate(itself, itself);

                    Global.Trash = links.Update(l2, l2, l1, l2);

```

```
286                     links.Delete(l1);
287
288                     transaction.Commit();
289                 }
290
291                 Global.Trash = links.Count();
292             }
293
294             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
295         }
296
297         /// <summary>
298         /// <para>
299         /// Tests that transaction damage.
300         /// </para>
301         /// <para></para>
302         /// </summary>
303         [Fact]
304         public static void TransactionDamage()
305         {
306             var itself = _constants.Itself;
307
308             var tempDatabaseFilename = Path.GetTempFileName();
309             var tempTransactionLogFilename = Path.GetTempFileName();
310
311             // Commit
312             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
313             using (var links = new UInt64Links(memoryAdapter))
314             {
315                 using (var transaction = memoryAdapter.BeginTransaction())
316                 {
317                     var l1 = links.CreateAndUpdate(itself, itself);
318                     var l2 = links.CreateAndUpdate(itself, itself);
319
320                     Global.Trash = links.Update(l2, l2, l1, l2);
321
322                     links.Delete(l1);
323
324                     transaction.Commit();
325                 }
326
327                 Global.Trash = links.Count();
328             }
329
330             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
331
332             // Damage database
333
334             FileHelpers.WriteFirst(tempTransactionLogFilename, new
                ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
335
336             // Try load damaged database
337             try
338             {
339                 // TODO: Fix
340                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
                    ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
341                 using (var links = new UInt64Links(memoryAdapter))
342                 {
343                     Global.Trash = links.Count();
344                 }
345             }
346             catch (NotSupportedException ex)
347             {
348                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
                    ↪ yet.");
349             }
350
351             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
                ↪ sactionLogFilename);
352
353             File.Delete(tempDatabaseFilename);
354             File.Delete(tempTransactionLogFilename);
355         }
356
357         /// <summary>
```

```
        /// <para>
        /// Tests that bug 1 test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void Bug1Test()
        {
            var tempDatabaseFilename = Path.GetTempFileName();
            var tempTransactionLogFilename = Path.GetTempFileName();

            var itself = _constants.Itself;

            // User Code Error (Autoreverted), some data saved
            try
            {
                ulong l1;
                ulong l2;

                using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                ↪  tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    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);
                }

                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                ↪  TransactionLogFilename);

                using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
                using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
                ↪  tempTransactionLogFilename))
                using (var links = new UInt64Links(memoryAdapter))
                {
                    using (var transaction = memoryAdapter.BeginTransaction())
                    {
                        l2 = links.Update(l2, l1);

                        links.Delete(l2);

                        ExceptionThrower();

                        transaction.Commit();
                    }

                    Global.Trash = links.Count();
                }
            }
            catch
            {
                Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
                ↪  TransactionLogFilename);
            }

            File.Delete(tempDatabaseFilename);
            File.Delete(tempTransactionLogFilename);
        }

        private static void ExceptionThrower() => throw new InvalidOperationException();

        /// <summary>
        /// <para>
        /// Tests that paths test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void PathsTest()
        {
            var source = _constants.SourcePart;
            var target = _constants.TargetPart;

```

```
433             using (var scope = new TempLinksTestScope())
434             {
435                 var links = scope.Links;
436                 var l1 = links.CreatePoint();
437                 var l2 = links.CreatePoint();
438 
439                 var r1 = links.GetByKeys(l1, source, target, source);
440                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
441             }
442         }
443 
444         /// <summary>
445         /// <para>
446         /// Tests that recursive string formatting test.
447         /// </para>
448         /// <para></para>
449         /// </summary>
450         [Fact]
451         public static void RecursiveStringFormattingTest()
452         {
453             using (var scope = new TempLinksTestScope(useSequences: true))
454             {
455                 var links = scope.Links;
456                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
457 
458                 var a = links.CreatePoint();
459                 var b = links.CreatePoint();
460                 var c = links.CreatePoint();
461 
462                 var ab = links.GetOrCreate(a, b);
463                 var cb = links.GetOrCreate(c, b);
464                 var ac = links.GetOrCreate(a, c);
465 
466                 a = links.Update(a, c, b);
467                 b = links.Update(b, a, c);
468                 c = links.Update(c, a, b);
469 
470                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
471                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
472                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
473 
474                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
                    ↪  "(5:(4:5 (6:5 4)) 6)");
475                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
                    ↪  "(6:(5:(4:5 6) 6) 4)");
476                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
                    ↪  "(4:(5:4 (6:5 4)) 6)");
477 
478                 // 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)"
479 
480                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
                    ↪  "{{5}{5}{4}{6}}");
481                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
                    ↪  "{{5}{6}{6}{4}}");
482                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
                    ↪  "{{4}{5}{4}{6}}");
483             }
484         }
485 
486         private static void DefaultFormatter(StringBuilder sb, ulong link)
487         {
488             sb.Append(link.ToString());
489         }
490 
491         #endregion
492 
493         #region Performance
494 
495         /*
496         public static void RunAllPerformanceTests()
497         {
498             try
499             {
500                 links.TestLinksInSteps();
501             }
502             catch (Exception ex)
503             {
```

```
504             ex.WriteToConsole();
505         }
506 
507         return;
508 
509         try
510         {
511             //ThreadPool.SetMaxThreads(2, 2);
512 
513             // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
    ↪  результат
514             // Также это дополнительно помогает в отладке
515             // Увеличивает вероятность попадания информации в кэши
516             for (var i = 0; i < 10; i++)
517             {
518                 //0 - 10 ГБ
519                 //Каждые 100 МБ срез цифр
520 
521                 //links.TestGetSourceFunction();
522                 //links.TestGetSourceFunctionInParallel();
523                 //links.TestGetTargetFunction();
524                 //links.TestGetTargetFunctionInParallel();
525                 links.Create64BillionLinks();
526 
527                 links.TestRandomSearchFixed();
528                 //links.Create64BillionLinksInParallel();
529                 links.TestEachFunction();
530                 //links.TestForeach();
531                 //links.TestParallelForeach();
532             }
533 
534             links.TestDeletionOfAllLinks();
535 
536         }
537         catch (Exception ex)
538         {
539             ex.WriteToConsole();
540         }
541     }*/
542 
543      /*
544     public static void TestLinksInSteps()
545     {
546         const long gibibyte = 1024 * 1024 * 1024;
547         const long mebibyte = 1024 * 1024;
548 
549         var totalLinksToCreate = gibibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
550         var linksStep = 102 * mebibyte /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
551 
552         var creationMeasurements = new List<TimeSpan>();
553         var searchMeasuremets = new List<TimeSpan>();
554         var deletionMeasurements = new List<TimeSpan>();
555 
556         GetBaseRandomLoopOverhead(linksStep);
557         GetBaseRandomLoopOverhead(linksStep);
558 
559         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
560 
561         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
562 
563         var loops = totalLinksToCreate / linksStep;
564 
565         for (int i = 0; i < loops; i++)
566         {
567             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
568             searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
569 
570             Console.Write("\rC + S {0}/{1}", i + 1, loops);
571         }
572 
573         ConsoleHelpers.Debug();
574 
575         for (int i = 0; i < loops; i++)
576         {
577             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
578 
579             Console.Write("\rD {0}/{1}", i + 1, loops);
580         }
```

```
581
582             ConsoleHelpers.Debug();
583
584             ConsoleHelpers.Debug("C S D");
585
586             for (int i = 0; i < loops; i++)
587             {
588                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
589             }
590
591             ConsoleHelpers.Debug("C S D (no overhead)");
592
593             for (int i = 0; i < loops; i++)
594             {
595                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
596             }
597
598             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
599         }
600
601         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
602         {
603             for (long i = 0; i < amountToCreate; i++)
604                 links.Create(0, 0);
605         }
606
607          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
608          {
609              return Measure(() =>
610              {
611                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
612                  ulong result = 0;
613                  for (long i = 0; i < loops; i++)
614                  {
615                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
616                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
617
618                      result += maxValue + source + target;
619                  }
620                  Global.Trash = result;
621              });
622          }
623           */
624
625          /// <summary>
626          /// <para>
627          /// Tests that get source test.
628          /// </para>
629          /// <para></para>
630          /// </summary>
631          [Fact(Skip = "performance test")]
632          public static void GetSourceTest()
633          {
634              using (var scope = new TempLinksTestScope())
635              {
636                  var links = scope.Links;
637                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
638
639                  ulong counter = 0;
640
641                  //var firstLink = links.First();
642                  // Создаём одну связь, из которой будет производить считывание
643                  var firstLink = links.Create();
644
645                  var sw = Stopwatch.StartNew();
646
647                  // Тестируем саму функцию
648                  for (ulong i = 0; i < Iterations; i++)
649                  {
650                      counter += links.GetSource(firstLink);
651                  }
652
653                  var elapsedTime = sw.Elapsed;
654
```

```
655                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
656
657                 // Удаляем связь, из которой производилось считывание
658                 links.Delete(firstLink);
659
660                 ConsoleHelpers.Debug(
661                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
662                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
663             }
664         }
665
666         /// <summary>
667         /// <para>
668         /// Tests that get source in parallel.
669         /// </para>
670         /// <para></para>
671         /// </summary>
672         [Fact(Skip = "performance test")]
673         public static void GetSourceInParallel()
674         {
675             using (var scope = new TempLinksTestScope())
676             {
677                 var links = scope.Links;
678                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
                    ↪  parallel.", Iterations);
679
680                 long counter = 0;
681
682                 //var firstLink = links.First();
683                 var firstLink = links.Create();
684
685                 var sw = Stopwatch.StartNew();
686
687                 // Тестируем саму функцию
688                 Parallel.For(0, Iterations, x =>
689                 {
690                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
691                     //Interlocked.Increment(ref counter);
692                 });
693
694                 var elapsedTime = sw.Elapsed;
695
696                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
697
698                 links.Delete(firstLink);
699
700                 ConsoleHelpers.Debug(
701                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per
                        ↪  second), counter result: {3}",
702                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
703             }
704         }
705
706         /// <summary>
707         /// <para>
708         /// Tests that test get target.
709         /// </para>
710         /// <para></para>
711         /// </summary>
712         [Fact(Skip = "performance test")]
713         public static void TestGetTarget()
714         {
715             using (var scope = new TempLinksTestScope())
716             {
717                 var links = scope.Links;
718                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
                    ↪  Iterations);
719
720                 ulong counter = 0;
721
722                 //var firstLink = links.First();
723                 var firstLink = links.Create();
724
725                 var sw = Stopwatch.StartNew();
726
727                 for (ulong i = 0; i < Iterations; i++)
728                 {
729                     counter += links.GetTarget(firstLink);
```

```
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪  second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that test get target in parallel.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "performance test")]
        public static void TestGetTargetInParallel()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
                ↪  parallel.", Iterations);

                long counter = 0;

                //var firstLink = links.First();
                var firstLink = links.Create();

                var sw = Stopwatch.StartNew();

                Parallel.For(0, Iterations, x =>
                {
                    Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
                    //Interlocked.Increment(ref counter);
                });

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

                links.Delete(firstLink);

                ConsoleHelpers.Debug(
                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
                    ↪  second), counter result: {3}",
                    Iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        // TODO: Заполнить базу данных перед тестом
        /*
        [Fact]
        public void TestRandomSearchFixed()
        {
            var tempFilename = Path.GetTempFileName();

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
            {
                long iterations = 64 * 1024 * 1024 /
    ↪  Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;

                ulong counter = 0;
                var maxLink = links.Total;

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var source =
    ↪  RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
```

```
                    var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

                    counter += links.Search(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
            }

            File.Delete(tempFilename);
        }*/

        /// <summary>
        /// <para>
        /// Tests that test random search all.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestRandomSearchAll()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ulong counter = 0;

                var maxLink = links.Count();

                var iterations = links.Count();

                ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
                ↪ links.Count());

                var sw = Stopwatch.StartNew();

                for (var i = iterations; i > 0; i--)
                {
                    var linksAddressRange = new
                    ↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);

                    var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
                    var target = RandomHelpers.Default.NextUInt64(linksAddressRange);

                    counter += links.SearchOrDefault(source, target);
                }

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
                ↪ Iterations per second), c: {3}",
                     iterations, elapsedTime, (long)iterationsPerSecond, counter);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that test each.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestEach()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

                var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);

                ConsoleHelpers.Debug("Testing Each function.");

```

```
877                 var sw = Stopwatch.StartNew();
878
879                 links.Each(counter.IncrementAndReturnTrue);
880
881                 var elapsedTime = sw.Elapsed;
882
883                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
884
885                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
                    ↪  links per second)",
886                     counter, elapsedTime, (long)linksPerSecond);
887             }
888         }
889
890         /*
891         [Fact]
892         public static void TestForeach()
893         {
894             var tempFilename = Path.GetTempFileName();
895
896             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
897             {
898                 ulong counter = 0;
899
900                 ConsoleHelpers.Debug("Testing foreach through links.");
901
902                 var sw = Stopwatch.StartNew();
903
904                 //foreach (var link in links)
905                 //{
906                 //    counter++;
907                 //}
908
909                 var elapsedTime = sw.Elapsed;
910
911                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
912
913                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
    ↪  links per second)", counter, elapsedTime, (long)linksPerSecond);
914             }
915
916             File.Delete(tempFilename);
917         }
918         */
919
920         /*
921         [Fact]
922         public static void TestParallelForeach()
923         {
924             var tempFilename = Path.GetTempFileName();
925
926             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪  DefaultLinksSizeStep))
927             {
928
929                 long counter = 0;
930
931                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
932
933                 var sw = Stopwatch.StartNew();
934
935                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
936                 //{
937                 //    Interlocked.Increment(ref counter);
938                 //});
939
940                 var elapsedTime = sw.Elapsed;
941
942                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
943
944                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
    ↪  {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
945             }
946
947             File.Delete(tempFilename);
948         }
949         */
950
951         /// <summary>
```

```
952         /// <para>
953         /// Tests that create 64 billion links.
954         /// </para>
955         /// <para></para>
956         /// </summary>
957         [Fact(Skip = "performance test")]
958         public static void Create64BillionLinks()
959         {
960             using (var scope = new TempLinksTestScope())
961             {
962                 var links = scope.Links;
963                 var linksBeforeTest = links.Count();
964 
965                 long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
966 
967                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
968 
969                 var elapsedTime = Performance.Measure(() =>
970                 {
971                     for (long i = 0; i < linksToCreate; i++)
972                     {
973                         links.Create();
974                     }
975                 });
976 
977                 var linksCreated = links.Count() - linksBeforeTest;
978                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
979 
980                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
981 
982                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
983                     (long)linksPerSecond);
984             }
985         }
986 
987         /// <summary>
988         /// <para>
989         /// Tests that create 64 billion links in parallel.
990         /// </para>
991         /// <para></para>
992         /// </summary>
993         [Fact(Skip = "performance test")]
994         public static void Create64BillionLinksInParallel()
995         {
996             using (var scope = new TempLinksTestScope())
997             {
998                 var links = scope.Links;
999                 var linksBeforeTest = links.Count();
1000 
1001                 var sw = Stopwatch.StartNew();
1002 
1003                 long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
1004 
1005                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
1006 
1007                 Parallel.For(0, linksToCreate, x => links.Create());
1008 
1009                 var elapsedTime = sw.Elapsed;
1010 
1011                 var linksCreated = links.Count() - linksBeforeTest;
1012                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
1013 
1014                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    ↪  linksCreated, elapsedTime,
1015                     (long)linksPerSecond);
1016             }
1017         }
1018 
1019         /// <summary>
1020         /// <para>
1021         /// Tests that test deletion of all links.
1022         /// </para>
1023         /// <para></para>
1024         /// </summary>
1025         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
1026         public static void TestDeletionOfAllLinks()
1027         {
1028             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          private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
            ↪  meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
            ↪  numberToAddressConverter)
68          {
69              var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
70              var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
                ↪  addressToNumberConverter, unicodeSymbolMarker);
71              var originalCharacter = 'H';
72              var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
73              var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                ↪  unicodeSymbolMarker);
74              var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
                ↪  numberToAddressConverter, unicodeSymbolCriterionMatcher);
75              var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
76              Assert.Equal(originalCharacter, resultingCharacter);
77          }
78
79          /// <summary>
80          /// <para>
81          /// Tests that string and unicode sequence converters test.
82          /// </para>
83          /// <para></para>
84          /// </summary>
85          [Fact]
86          public static void StringAndUnicodeSequenceConvertersTest()
87          {
88              using (var scope = new TempLinksTestScope())
89              {
90                  var links = scope.Links;
91
92                  var itself = links.Constants.Itself;
93
94                  var meaningRoot = links.CreatePoint();
95                  var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
96                  var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
97                  var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
98                  var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
99                  var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
100
101                 var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
102                 var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
103                 var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
104
105                 var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
106                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
107                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
108                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
109                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
110                 var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
111                 var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
112                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
113
114                 var stringToUnicodeSequenceConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
115
```

```
116                     var originalString = "Hello";
117
118                     var unicodeSequenceLink =
                        ↪  stringToUnicodeSequenceConverter.Convert(originalString);
119
120                     var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                        ↪  unicodeSymbolMarker);
121                     var unicodeSymbolToCharConverter = new
                        ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                        ↪  unicodeSymbolCriterionMatcher);
122
123                     var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                        ↪  unicodeSequenceMarker);
124
125                     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                        ↪  unicodeSymbolCriterionMatcher.IsMatched);
126
127                     var unicodeSequenceToStringConverter = new
                        ↪  UnicodeSequenceToStringConverter<ulong>(links,
                        ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                        ↪  unicodeSymbolToCharConverter);
128
129                     var resultingString =
                        ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
130
131                     Assert.Equal(originalString, resultingString);
132                 }
133             }
134         }
135     }
```

Index

./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs, 139
./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs, 140
./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 141
./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs, 142
./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs, 143
./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs, 144
./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.cs, 147
./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs, 149